

PHP

Endlich objektorientiert

OO- und UML-Praxisbuch: vom Anfänger zum Fortgeschrittenen

Dr. Frank Dopatka

Frank Dopatka

PHP – Endlich objektorientiert

OO- und UML-Praxisbuch: vom Anfänger zum Fortgeschrittenen

Frank Dopatka

PHP – Endlich objektorientiert

OO- und UML-Praxisbuch: vom Anfänger zum Fortgeschrittenen

entwickler.press

Frank Dopatka
PHP – Endlich objektorientiert
OO- und UML-Praxisbuch: vom Anfänger zum Fortgeschrittenen
ISBN: 978-3-86802-039-7

Ein Imprint der Software & Support Verlag GmbH

Bibliografische Information Der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.ddb.de abrufbar.

Ihr Kontakt zum Verlag und Lektorat:
Software & Support Verlag GmbH
entwickler.press
Geleitsstraße 14
60599 Frankfurt
Tel.: +49 (0)69 630089 0
Fax: +49 (0)69 630089 89
lektorat@entwickler-press.de
http://www.entwickler-press.de

Lektorat: Sebastian Burkart
Korrektorat: Katharina Klassen und Frauke Pesch
Satz: mediaService, Siegen
Belichtung, Druck & Bindung: M.P. Media-Print Informationstechnologie GmbH, Paderborn

Inhaltsverzeichnis

1 Eine Sprache wird erwachsen

Die Zeiten, in denen man die von dem Dänen Rasmus Lerdorf entwickelte Skriptsprache PHP als „Personal Home Page Tools“ bezeichnete, um dynamische HTML-Tabellen aus einer kleinen MySQL-Datenbank, Kontaktformulare oder einzelne Onlineangebote zu erstellen, sind vorüber. Mit der Version 4.0 im Jahr 2000 bis hin zur Version 4.4.9 im Jahr 2008 wurde die Entwicklung von PHP zu einer erwachsenen Sprache mit eingebauter Datenbankunterstützung vorangetrieben, die sich weder vor Konkurrenten wie Active Server Pages (ASP) von Microsoft, noch vor JavaServer Pages (JSP) von Sun Microsystems verstecken muss.

PHP hat in diesen Jahren den Ruf erlangt, leicht erlernbar zu sein und effiziente Internetlösungen hervorzubringen, die gerade im Umfeld der Web-2.0-Euphorie und der Vernetzung der Haushalte mit schneller DSL-Technologie sehr gefragt sind. Viele Provider bieten bereits LAMP-Server (Linux, Apache, MySQL, PHP) für Privatkunden an. Der Aufwand der Administration hält sich im Vergleich zu den Microsoft-Lösungen und den Java-Containern in Grenzen. So hat sich mit den Jahren eine erfolgreiche prozedurale, in C entwickelte Skriptsprache entwickelt, die sogar ihren Namen weiterentwickelt hat, der nun „Hypertext Preprocessor“ lautet. Dieser Name entstand, da der PHP-Interpreter den geschriebenen Quellcode in Form von herkömmlichen Textdateien vorverarbeitet, bevor der Datenstrom zum Webserver weitergeleitet wird.

Während PHP 4 nun ab dem Jahr 2000 bis heute seinen Siegeszug antritt, wurde bereits 2004 die fünfte Version von PHP veröffentlicht. Mit dieser Version wurden objektorientierte Ansätze in die Sprache integriert sowie eine Unterstützung der Verarbeitung von XML-Daten. Die neue PHP-Version wurde über einen längeren Zeitraum kaum wahrgenommen. Aus welchen Gründen soll man sich kompliziertere, objektorientierte Konzepte aneignen? Entwickler von vorwiegend kleinen PHP-Skripten für die eigene Homepage waren sehr zufrieden mit der herkömmlichen Art, in PHP zu programmieren.

Interessanterweise wird in den Stellengesuchen der letzten Monate verstärkt die Anforderung an Entwickler gestellt, „objektorientiert in PHP“ programmieren zu können. Außerdem werden die Begriffe der Geschäftsprozeßanalyse, der objektorientierten Analyse und des objektorientierten Designs (GPA, OOA und OOD) mit PHP in Verbindung gebracht. Ebenso ist die Nachfrage nach Schulungen im Bereich der service- und objektorientierten Konzepten seit einem Jahr stark gestiegen. Der Fokus der Entscheider wird seit einiger Zeit auf PHP 5 gelegt. Hier ist zunächst die Frage zu stellen, wie dieser Sinneswandel zu erklären ist?

1.1 Prozedurale Programmierung versus Objektorientierung

Die bis zu PHP 4 vorherrschende prozedurale Programmierweise besteht darin, eine Problemstellung in kleinere Unterprobleme nach dem Motto „Teile und Herrsche" (Divide and Conquer) aufzuteilen. Die Idee besteht darin, dass die einzelnen Teilprobleme unabhängig betrachtet werden können und leichter lösbar sind. Für jedes Unterproblem wurde dann eine Funktion geschrieben, die intern wiederum andere Funktionen aufrufen kann, um ihren Zweck zu erfüllen. Zusätzlich kann jede Funktion den Sprachumfang, also einzelne Befehle von PHP benutzen, um zur Lösung zu gelangen.

Eine Funktion könnte beispielsweise *Login* lauten und als Parameter den Benutzernamen und das Kennwort erhalten. Diese Daten wurden zuvor in ein HTML-Formular eingegeben und an ein PHP-Skript weitergeleitet. Diese Funktion *Login* wird die Unterfunktionen *DBconnect* und *DBlesen* enthalten, um serverseitig eine Verbindung zur Datenbank herzustellen und zu prüfen, ob ein Benutzer mit diesem Benutzernamen überhaupt existiert. Ist das der Fall, so wird das richtige Kennwort aus der Datenbank ausgelesen und mit dem Kennwort aus dem HTML-Formular mittels PHP-Befehlen verglichen. Sind beide gleich, so gibt die Funktion *Login* den Wert *OK* zurück.

Zu dem prozeduralen Programmierstil gehört außerdem, dass man einerseits Daten in einer Datenbank besitzt und andererseits Funktionen, die mit diesen Daten arbeiten. Die Funktionalität des Programms wird bei der prozeduralen Programmierung von den Daten getrennt. Die Daten durchlaufen die Funktionen und werden von ihnen interpretiert und verarbeitet.

Die Denkweise der prozeduralen Programmierung wird von den Anhängern der Objektorientierung oft als „veraltet" angesehen. Man programmiert heutzutage nicht mehr auf diese Art. Dieser Aussage widerspricht jedoch der Erfolg der bisherigen Vorgehensweise bei der Erstellung von PHP-Programmen.

In der aktuellen PHP-5-Version sind beide Vorgehensweisen erlaubt. Einem Einsteiger in die PHP-Programmierung wird dadurch nicht gerade geholfen, dass er noch zwischen zwei verschiedenen Denkweisen unterscheiden muss.

Doch wodurch unterscheidet sich die Objektorientierung in ihrer Denkweise? In der Objektorientierung werden die zu verarbeitenden Daten anhand ihrer Eigenschaften und der möglichen Operationen klassifiziert. Man hat also das Objekt *Kunde*. Ein Kunde besitzt seine eigenen Daten wie *Name*, *Vorname* usw. und zusätzlich dazu einen Satz von Funktionen. Man kann einen Kunden beispielsweise nach seinem Namen oder nach seinen Rechnungen fragen.

Im Vergleich zu den anderen Denkweisen wird von der Objektorientierung gesagt, dass sie menschliche Organisationsmethoden aus der realen Welt besser nachbilden kann. Die Entwicklung der Software orientiert sich also mehr an den menschlichen Abläufen als an einer Maschinensprache.

Aus technischer Sicht werden die Daten der Objekte dennoch wieder separat und getrennt von ihrer Funktionalität in den Datenbanken verwaltet. Langfristig versucht man, die relationalen Datenbanken durch objektorientierte Datenbanken zu ersetzen, in

denen man direkt die Objekte verwaltet. Heutzutage programmiert man ein Datenzugriffsobjekt, das eine Verbindung zur Datenbank repräsentiert. Das Datenzugriffsobjekt fragt die Objekte an, welche gerade von Interesse sind. Das Datenzugriffsobjekt sucht sich die passenden Daten aus der relationalen Datenbank, baut die Objekte aus diesen Daten zusammen und gibt diese Objekte dann als Ergebnis der Anfrage zurück. Die Anwendung arbeitet dann nur mit diesen Objekten, indem die in den Objekten integrierte Funktionalität verwendet wird. Über das Zugriffsobjekt werden die Daten auch wieder gespeichert.

Es stellt sich die Frage, wann die Objektorientierung der prozeduralen Programmierung vorzuziehen ist? Man kann sagen, dass bei kleinen Projekten bis zu 10 000 Codezeilen bzw. bis zu zwei Personenjahren keine besondere formale Planung notwendig ist. Bei diesen Projekten kann eine einzelne Person den Überblick wahren und die vollständige Realisierung selbst vornehmen. Das ist typisch für freiberufliche PHP-Einzelentwickler, die in eigener Regie kleine Lösungen für ihre Kunden erstellen. Es sind weder Entwicklerteams, noch komplexe Werkzeuge und Editoren notwendig. In diesem Umfeld hat sich PHP 4 bereits etabliert und bewährt.

Durch die hohe Verbreitung und den guten Ruf von PHP wird die Skriptsprache jedoch in letzter Zeit verstärkt in größeren Projekten eingesetzt. Das bedeutet, das sowohl auf der Seite des Kunden als auch auf der Seite der Entwickler eine Vielzahl von Personen am Projekt beteiligt ist. Meistens sind sogar die Anforderungen an das Projekt im Vorfeld nur grob oder gar nicht bekannt. In diesem Fall werden eine Geschäftsprozeßanalyse und agile Vorgehensmethoden der Softwareentwicklung eingesetzt. Auch die Objektorientierung besteht nicht nur aus der objektorientierten Programmierung (OOP). Im Anschluss an die Geschäftsprozeßanalyse erfolgt in großen Projekten in einem iterativ-inkrementellen Prozess eine objektorientierte Analyse und ein Design der Lösung, das in einer objektorientierten Programmierung mit PHP mündet.

Als gemeinsame Sprache für Kunden, Analytiker und Entwickler während der OOA und OOD hat sich in den letzten Jahren die Unified Modeling Language (UML) durchgesetzt. Die UML bietet auch eine wohl definierte Vorgehensweise, um von einem Kundenwunsch zu einem Softwareartefakt zu gelangen. Es ist also eine genaue Planung, sowohl der Kosten als auch sämtlicher Ressourcen erforderlich. Auch die Werkzeuge, die zum Einsatz kommen, unterscheiden sich grundlegend von einem herkömmlichen Editor. So sind Tools zur Projektverwaltung, Quellcodeversionierung, zur Dokumentation sowie zur Durchführung von automatisierten Funktions- und Integrationstests notwendig geworden.

Für einen unerfahrenen Entwickler, der sich die durchaus interessante Sprache PHP erstmalig ansehen will, mögen diese Ausführungen zunächst abschreckend sein. Das führt direkt zur Fragestellung nach der Zielgruppe dieses Buchs sowie zu dessen Aufbau.

1.2 Zielgruppe dieses Buchs

Nach den komplexen Ausführungen ist zunächst beruhigend, nochmals zu betonen, dass die prozedurale Programmierung auch weiterhin mit PHP 5 möglich ist. Eine Anleitung zur Erstellung dynamischer Webauftritte mit PHP ist in diesem Buch im zweiten

Kapitel enthalten. Dadurch finden auch die Entwickler einen Einstieg, die im Vorfeld noch keine PHP-Erfahrung besitzen und lediglich keine Vorhaben realisieren wollen. Statt einzelne Befehle lediglich zu definieren, wird in diesem Buch praxisnah die Erstellung von PHP-Skripten beschrieben.

Der Überblick über die grundlegenden Möglichkeiten von PHP ist jedoch bewusst kurz gehalten. Eine Erfahrung in einer anderen Programmiersprache und/oder einer anderen Internetskriptsprache ist an dieser Stelle sicherlich vorteilhaft. Zusätzlich ist die Erstellung von kleinen PHP-Lösungen bereits in Internetforen wie *http://www.php.de* ausreichend beschrieben.

Der Fokus dieses Buchs liegt auf Entwicklern und Projektleitern, die bereits erste Erfahrungen mit der traditionellen PHP-Programmierung gesammelt haben bzw. die in Zukunft größere PHP-Projekte planen. Erfahrungsgemäß sind diese Entwickler nur wenig mit den Konzepten der Objektorientierung und der UML vertraut und wollen/müssen ihren Umgang mit PHP und der neuen Denkweise professionalisieren. Das ist die zentrale Zielgruppe dieses Buchs.

Projektleiter und Freiberufler mit größeren Projekten im PHP-Umfeld werden mithilfe dieses Buchs eine praxisnahe Methodik kennen lernen, wie sie Kundenwünsche ermitteln, erfassen und strukturieren können. Das bildet die Grundlage für eine Umsetzung in objektorientiertem PHP-Quellcode.

1.3 Gliederung und Vorgehensweise

Im folgenden Kapitel wird im ersten Schritt die grundlegende Syntax der Sprache PHP und deren Einsatz in Verbindung mit HTML beschrieben. Neben den wichtigsten Befehlen wird gezeigt, wie man Funktionen schreibt, PHP-Dateien strukturiert und Verzweigungen sowie Schleifen in PHP realisiert. Dieses Kapitel ist insbesondere für Einsteiger gedacht, die bislang noch keine PHP-Skripte selbst verfasst haben. Außerdem wird gezeigt, wie Sie ausgefüllte HTML-Formulare mit PHP verarbeiten, einen Warenkorb mit einer Session verwalten, einen Zugriff auf eine MySQL-Datenbank realisieren und auf einen E-Mail-Server zugreifen. Dieses Kapitel verdeutlicht die traditionelle prozedurale Programmierung mit PHP.

Das dritte Kapitel steigt in die objektorientierte Denkweise ein, die zunächst unabhängig von PHP ist. Die Objektorientierung besitzt eine eigene Sprache, deren Begriffe und Vokabeln in diesem Kapitel anwendungsbezogen beschrieben werden. Dabei erlangen Sie einen Einstieg in die weit verbreitete UML-Notation, die Sie und alle Projektbeteiligten von der Idee bis zur Umsetzung begleiten wird. Es wird dargestellt, wie man mithilfe der UML von einer gewünschten Funktionalität eines Kunden über eine Geschäftsprozeßanalyse, eine fachliche Modellierung im Rahmen der objektorientierten Analyse zu einem objektorientierten Design gelangt.

Das objektorientierte Design repräsentiert die technische Modellierung, die später unter Verwendung von PHP umgesetzt werden soll. In Verbindung mit der Objektorientierung sind die Begriffe der testgetriebenen Entwicklung (Test-driven Development – TDD) und der featuregetriebenen Entwicklung (Feature-driven Development – FDD) entstanden,

die zum Ende des Kapitels vorgestellt werden. Das dritte Kapitel schließt mit der Vorstellung bekannter Design Patterns, die bewährte Schablonen in der objektorientierten Programmierung darstellen.

Im vierten Kapitel wird nun gezeigt, wie man die vorgestellten objektorientierten Konzepte mit PHP 5 umsetzen kann. Dabei wird jeder Begriff in UML dargestellt und der Realisierung in PHP gegenübergestellt. Sie werden beispielsweise im dritten Kapitel lernen, was Vererbung bedeutet. Sowohl im dritten als auch im vierten Kapitel wird die Vererbung in der UML-Notation dargestellt. Im vierten Kapitel sehen Sie dann, wie man eine Vererbung in PHP umsetzt. Zusätzlich dazu werden typische Konstrukte erstellt, die anwendungsübergreifend weiter verwendet werden können. Dazu gehört die Erstellung eines Datenzugriffsobjekts auf eine MySQL-Datenbank, die objektorientierte Umsetzung einer XML-Verarbeitung bis hin zur Programmierung eines Web Service in PHP. Zusätzlich wird skizziert, wie die wichtigsten Design Patterns mit PHP 5 umgesetzt werden können.

Das fünfte Kapitel stellt ein größeres Fallbeispiel vor, das den Einsatz der Objektorientierung in größeren Projekten skizziert. Dazu gehört ein kompletter Projektverlauf von einer Kundenidee über die Geschäftsprozeßanalyse, objektorientierte Analyse und Design bis hin zu der fertigen PHP-Anwendung einer Verwaltung von persönlichen Aktienbeständen.

Im Anschluss daran steht die sinnvolle Strukturierung eines PHP-Projekts im Vordergrund. Dazu gehört die Umsetzung der im dritten Kapitel beschriebenen testgetriebenen Entwicklung unter Verwendung des Tools PHPUnit sowie Coding-Standards, die einen guten PHP-Quelltext auszeichnen. Abschließend werden zahlreiche Tools vorgestellt, die bei der objektorientierten Entwicklung mit PHP hilfreich sind. Einem einzelnen Entwickler genügt vielleicht ein Texteditor mit Syntax-Highlighting. Für komplexere Projekte sind jedoch komplexere Werkzeuge notwendig.

1.4 Verwendete Software

Der in diesem Buch verwendete PHP-Interpreter wurde als Teil des XAMPP-Pakets (beliebiges Betriebssystem: X, Apache, MySQL, Perl und PHP) in der Version 1.7.2 installiert. Zu dem Paket gehört der Webserver Apache 2.2.12, der Datenbankserver MySQL 5.1.37, PHP in der Version 5.3.0 sowie das in PHP programmierte Tool zur Verwaltung des Datenbankservers phpMyAdmin in der Version 3.2.0.1. Als Betriebssystem wurde Windows XP Professional mit ServicePack 3 verwendet, sodass ein WAMP-Server (Windows, Apache, MySQL, PHP) entsteht.

Die UML-Diagramme, die insbesondere in den Abbildungen im dritten Kapitel zu sehen sind, wurden mit Microsoft Visio 2003 Professional gezeichnet. Auf die Anwendung der UML-Schablonen von Visio wurde verzichtet; die Diagramme wurden lediglich aus elementaren Zeichnungselementen (Vierecke, Linien, Text usw.) erstellt.

2 Die Sprache PHP: Prozedural

In diesem Kapitel wird die grundlegende Syntax von PHP vorgestellt. Es richtet sich auch an Programmieranfänger, die bislang noch keinen Bezug zu einer Programmiersprache besitzen. Dieses Kapitel zeigt die wesentlichen Merkmale jeder prozeduralen Programmiersprache auf, z. B. Verzweigungen und Schleifen. Wenn Sie diese Merkmale verstanden haben, werden Sie kein Problem haben, sich in eine andere prozedurale Sprache wie JavaScript, Microsoft Visual Basic 6 oder C einzuarbeiten.

Außerdem wird gezeigt, wie Sie Ihre Homepage um PHP-Funktionalität anreichern können. Interessant ist dieses Kapitel besonders dann, wenn Sie noch keinerlei Erfahrungen mit der Sprache besitzen, wenn Sie eine bestehende Internetpräsentation um einzelne PHP-Elemente erweitern wollen oder lediglich kleine Projekte in PHP realisieren wollen.

Im Gegensatz zu einer Programmiersprache, mit der Sie ausführbare Dateien erzeugen, die auf Ihrem Computer installiert und gestartet werden, handelt es sich bei PHP um eine „Internetsprache". Im Gegensatz zur clientseitigen Skriptsprache JavaScript wird PHP auf dem Server ausgeführt. PHP kann daher mit ASP und JSP verglichen werden, da bei allen drei Sprachen die HTML-Dateien, die auf dem Webserver liegen, mit fremdem Quellcode angereichert werden. Das bedeutet, dass in einer HTML-Datei, die ja zusätzlich noch JavaScript für die clientseitige Ausführung und CSS-Anweisungen (Cascading Stylesheets) für die Beschreibung des Designs der Homepage enthalten kann, mit einer weiteren Sprache versehen wird. Zusätzlich dazu muss die *.html*-Datei in *.php* umbenannt werden.

Wenn nun ein Internetbrowser als Client auf die PHP-Datei über den Webserver Apache zugreifen will, wird diese zuerst durch den PHP-Interpreter eingelesen. Die Ausgabe der PHP-Datei wird dann an den Webserver weitergegeben und dann an den Browser gesendet. Greift PHP dann noch auf die MySQL-Datenbank zu, so wird auch noch die Sprache SQL (Structured Query Language) in die Datei integriert. Es gilt also, den Überblick über diese Sprachen zu behalten. Dieses Buch soll dabei durch ein strukturiertes Vorgehen unterstützen.

2.1 Grundlegende Syntax

Als erster Schritt für das Erlernen einer neuen Programmiersprache steht das „Hallo Welt"-Programm. Das hat zum Ziel, eine erste Ausgabe als ersten Erfolg zu erreichen. Nach der Installation des XAMPP-Pakets befindet sich XAMPP standardmäßig unter MS Windows im Verzeichnis *C:\Programme\XAMPP*. Dort wird das Unterverzeichnis *htdocs* angelegt. Das ist das Basisverzeichnis für Ihre selbstgeschriebenen Dateien. Erstellen Sie in einem Texteditor im *htdocs*-Verzeichnis die Datei *hallo.html* mit folgendem Inhalt:

```
<html>
  <head>
    <title>Hallo</title>
  </head>
  <body>
    <h1>Hallo</h1>
  </body>
</html>
```

Listing 2.1: Die erste Datei hallo.html im htdocs-Verzeichnis des XAMPP-Servers

Um die Datei nun über den Webserver aufzurufen, müssen Sie diesen zuerst über das XAMPP Control Panel starten. Klicken Sie dazu auf die *Start*-Schaltfläche des Webservers (Abb. 2.1).

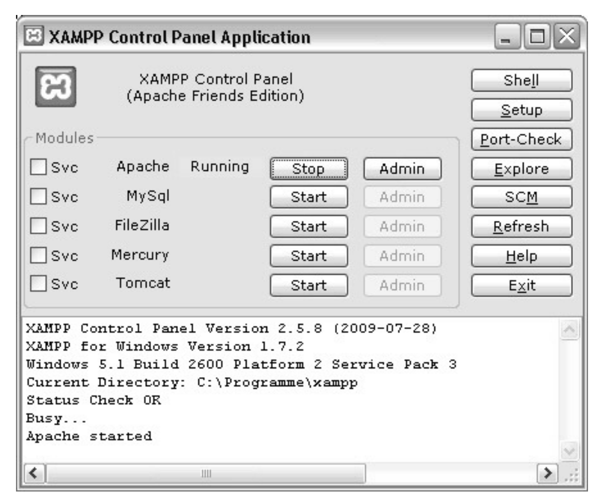


Abbildung 2.1: Das XAMPP Control Panel mit gestartetem Apache-Webserver

Öffnen Sie nun Ihren Internetbrowser und geben Sie *http://localhost/hallo.html* ein. Sie sehen jetzt ein „Hallo“ im Titel des Browsers sowie als Homepage. Damit haben Sie aber noch nicht den PHP-Interpreter verwendet, da es sich um eine HTML-Datei handelt. Wird eine HTTP-Anfrage auf eine HTML-Datei von einem Client auf einen Webserver gestellt, so wird diese Datei vom Webserver geöffnet, der Inhalt ausgelesen, in das HTTP-Protokoll verpackt und unmittelbar an den Client zurück gesendet (Abb. 2.2).

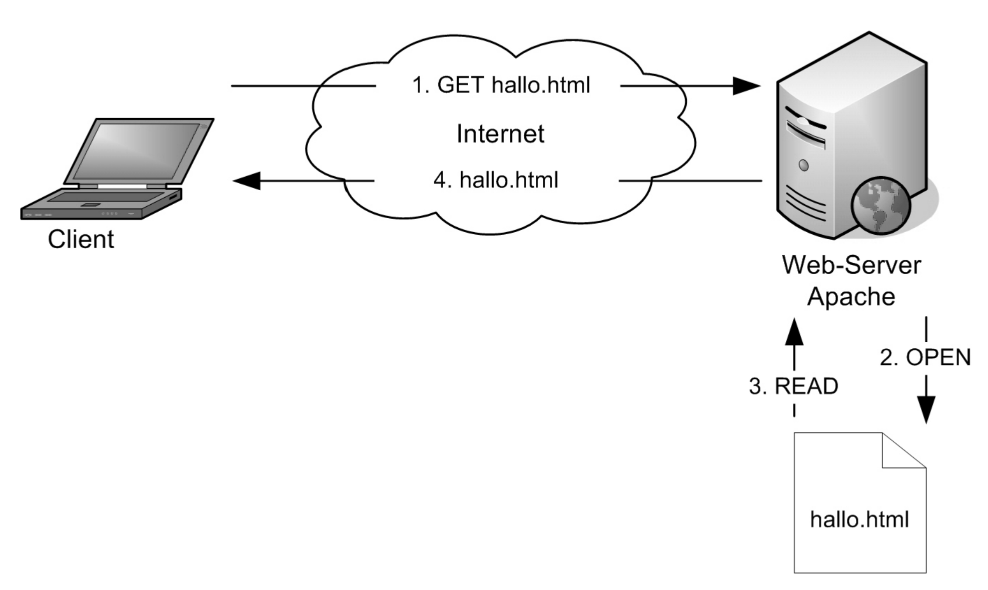


Abbildung 2.2: Aufruf einer HTML-Datei

Um Ihre erste PHP-Datei zu erstellen, erzeugen Sie in dem *htdocs*-Verzeichnis mit Ihrem Texteditor eine Datei mit dem Namen *hallo.php*:

```
<?php $wert='Hallo'; ?>
<html>
  <head>
    <title><?php echo $wert?></title>
  </head>
  <body>
    <h1><?php echo $wert?></h1>
  </body>
</html>
```

Listing 2.2: Das erste PHP-Skript hallo.php

Die PHP-Datei definiert zunächst eine Variable *wert* und belegt sie mit der Zeichenkette „Hallo“. Die Variable ist in der ganzen PHP-Datei gültig und wird an zwei Stellen im HTML-Code ausgegeben.

Die Ausgabe im Internetbrowser unterscheidet sich in keiner Weise von der Ausgabe der HTML-Datei. Wenn Sie sich im Browser den Quellcode der Datei ansehen, werden Sie ebenfalls keinen Unterschied feststellen. Was geschieht also bei der Verarbeitung der PHP-Datei?

Wird eine HTTP-Anfrage auf eine PHP-Datei gestellt, so öffnet der Webserver diese Datei und gibt sie an den PHP-Interpreter weiter. Jeder Bereich zwischen den Marken

<?php ?> wird daraufhin vom Interpreter als PHP-Code analysiert. Dieser Code kann wiederum eine HTML-Ausgabe erzeugen, die dann an den Webserver weitergegeben wird. Der PHP-Code selbst gelangt dabei nicht zum Client, lediglich dessen Ausgabe. Dadurch bleibt der PHP-Quellcode für den Client unsichtbar. Ebenso wird jeder Text außerhalb der Marken *<?php ?>* unmittelbar an den Webserver weitergeleitet. Das Prinzip der Verarbeitung einer PHP-Datei ist in Abbildung 2.3 skizziert.

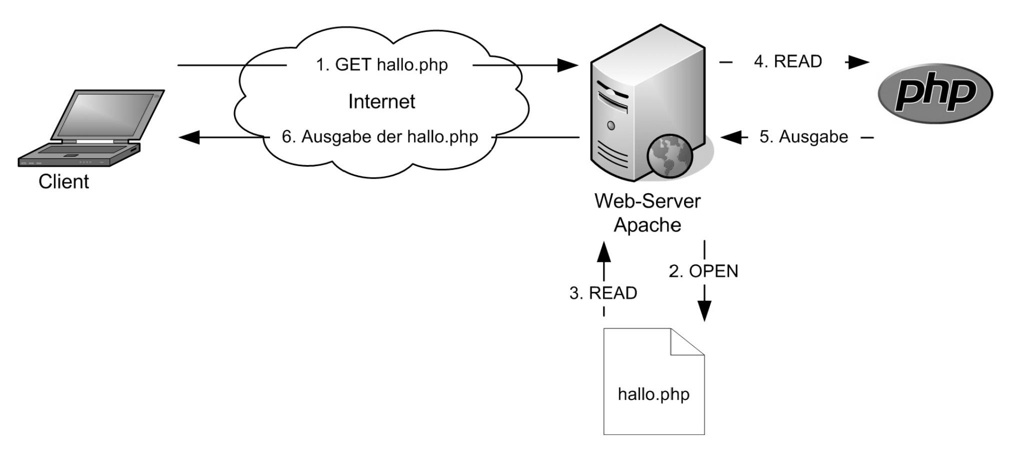


Abbildung 2.3: Aufruf einer PHP-Datei

Wenn Sie dieses Beispiel erfolgreich getestet haben, sind Sie erfahrungsgemäß sehr gespannt, welche Befehle die Sprache PHP bietet und auf welche Arten sich die Erzeugung von HTML-Code mit PHP manipulieren lässt.

2.1.1 Von Kommentaren, Variablen und Operanden

Als grundlegendes Element einer Programmiersprache ist die Definition von Konstanten und Variablen zu nennen. Dazu ist noch von Interesse, welche Datentypen eine Sprache anbietet und mit welchen vorgegebenen Operanden ein Programmierer Rechenoperationen ausführen kann.

Kommentare

Die Kommentare in PHP entsprechen in ihrer Syntax den Kommentaren von Java oder C++. Man unterscheidet zwischen einem Zeilenkommentar, der mit den Zeichen // beginnt und den Rest der Zeile als Kommentar markiert. Dort können dann beliebige Zeichen platziert werden, die keinen Einfluss auf die Verarbeitung haben. Wird der Kommentar vor einer Anweisung platziert, so wird diese Anweisung nicht ausgeführt; sie ist „auskommentiert“.

Die zweite Art der Kommentierung bezieht sich je nach Anwendung auf mehrere Zeilen. Sie beginnt mit /* und endet mit den Zeichen */. In Listing 2.3 wird von beiden Arten der Kommentierung Gebrauch gemacht.

Konstanten

Eine Konstante enthält wie auch eine Variable einen Wert. Im Unterschied zu einer Variablen wird eine Konstante einmalig gesetzt. Sie kann im Anschluss ausgelesen, aber nicht verändert werden. In PHP wird eine Konstante über den Befehl *define* erzeugt, der zwei Parameter erhält. Der erste Parameter erhält den Namen der Konstante und der zweite deren Inhalt:

```
<?php
  /*
    In diesem Skript wird eine Konstante definiert und im HTML-Teil
    des Skriptes verwendet.
    Autor: Dr. Frank Dopatka
  */
  define('WERT_KONST', 'Meine tolle Homepage'); // hier ist die Def.
?>
<html>
  <head>
    <title><?php echo WERT_KONST?></title>
  </head>
  <body>
    <h1><?php echo WERT_KONST?></h1>
  </body>
</html>
```

Listing 2.3: Definition einer Konstanten

Profitipp

Um Konstanten leicht im Quellcode zu erkennen, schreibt man sie am besten stets komplett in Großbuchstaben.

Variablen

Im Gegensatz zu einer Konstanten beginnt eine Variable immer mit dem $-Symbol, das von einem PHP-Anfänger leicht vergessen wird. Die Zuweisung eines Werts erfolgt direkt über den =-Operator. Listing 2.4 zeigt gültige Deklarationen von Variablen:

```
<?php
  $a=4; $b=1; $c=8.3; $d="Hallo"; $e='Hallo';
  $f=TRUE; $g=FALSE; $h=false;
  $i='<table><tr><td>Spalte1</td><td>Spalte2</td></tr></table>'
?>
```

Listing 2.4: Deklaration und Zuweisung von Variablen

Bei den Variablen *$a* und *$b* handelt es sich um Ganzzahlen (Integer), bei *$c* um eine Fließkommazahl (Double) und bei *$d* und *$e* um Zeichenketten. Zeichenketten können durch einfache[1] oder doppelte Anführungszeichen definiert werden. Die Variablen *$f*, *$g* und *$h* werden mit Wahrheitswerten (Boolean) belegt. Eine Ausgabe mittels *<?php echo $f?>* liefert 1, eine Ausgabe von *$g* und *$h* liefert keine Rückgabe.

Ein großer Anwendungsbereich von PHP liegt darin, Teile des HTML-Codes in Zeichenkettenvariablen auszulagern. Das ist bei *$i* exemplarisch geschehen. Wird *$i* ausgegeben, wird an der Stelle der Ausgabe eine HTML-Tabelle dynamisch in den HTML-Quelltext hinzugefügt. Auf diese Weise lässt sich die Erzeugung von HTML-Ausgabe serverseitig steuern.

Profitipp

Zeichenkettenoperationen mit einfachen Anführungszeichen werden um ca. 15 % schneller vom PHP-Interpreter bearbeitet. Bei hoher Last auf dem Webserver ist bei deren Verwendung ein Performancegewinn zu erreichen.

Eine häufige Fehlerquelle beim Debugging liegt darin, dass PHP zwischen Groß- und Kleinschreibung unterscheidet. Zusätzlich dazu kann man eine Variable abfragen, die zuvor keinen Wert erhalten hat. Der Quellcode in Listing 2.5 ergibt für viele PHP-Anfänger überraschend keine sichtbare Ausgabe im Browser. Das ist auf die fehlerhafte Groß-/Kleinschreibung zurückzuführen:

```
<?php $Wert=4; ?>
<html>
  <body>
    <?php echo $wert?><br>
  </body>
</html>
```

Listing 2.5: Verwendung einer nicht deklarierten Variable

Datentypen

Im Gegensatz zu anderen Sprachen verfügt PHP über eine übersichtliche Anzahl von Datentypen. Auffallend ist jedoch, dass man den Datentyp nicht bei der Erzeugung einer Variablen angibt, sondern direkt eine Wertzuweisung vornimmt. Man spricht bei PHP über eine untypisierte Sprache, die die folgenden Datentypen aus dem Kontext der Wertzuweisung erkennen kann:

- Integer
  - Der Wertebereich der Ganzzahlen liegt von -2.147.482.648 bis +2.147.482.647, also von (-231-1) bis 231. Das entspricht einer Größe von 4 Byte.
  - Bei einem Überlauf wandelt PHP den Typ automatisch nach Double um.

[1] Das einfache Anführungszeichen liegt bei deutschem Tastaturlayout auf der #-Taste und wird mit *STRG+#* aktiviert.

- Typische Deklarationen sind:
$a=1234;
$b=-123;
$c=0123; // Oktalzahl, die dem Wert 83 dezimal entspricht
$d=0x1A; // Hexadezimalzahl, die dem Wert 26 dezimal entspricht

- Double, auch Float genannt
  - Der Wertebereich der Fließkommazahlen liegt bei ca. 1.7E-308 bis ca. 1.7E+308, also von -21024-1 bis 21024. Das entspricht einer Größe von 8 Byte.
  - Die Genauigkeit beträgt 14 Nachkomma-Stellen.
  - Typische Deklarationen sind:
$a=1.234;
$b=1.2e3;
$c=7E-10;

- String
  - Eine Zeichenkette, deren Größe nur durch den bereit gestellten Speicher für PHP beschränkt wird.
  - PHP unterstützt bis inklusive Version 5 den internationalen Unicode-Zeichensatz nicht. Ein Zeichen besteht also nur aus einem Byte.
  - Folgende Sonderzeichen können direkt in einer Zeichenkette platziert werden:
n entspricht einem Zeilenvorschub (ASCII-Wert 10)
r entspricht einem Wagenrücklauf (ASCII-Wert 13)
t entspricht einem horizontalen Tabulator (ASCII-Wert 9)
v entspricht einem vertikalen Tabulator seit PHP 5.2.5 (ASCII-Wert 11)
f entspricht einem Seitenvorschub seit PHP 5.2.5 (ASCII-Wert 12)
\\\\ entspricht einem Backslash
$ entspricht dem Dollar-Zeichen
\\" entspricht einem doppelten Anführungszeichen

- Boolean
  - Ein Boolean-Ausdruck ist ein Wahrheitswert, der entweder *TRUE* (wahr) oder *FALSE* (falsch) sein kann.
  - Bei der Konvertierung eines anderen Datentyps zum Typ *Boolean* gelten die folgenden Werte als *FALSE*:
- Ein Integer, der *0* beinhaltet.
- Ein Double, der *0.0* beinhaltet.
- Eine leere Zeichenkette sowie die Zeichenkette *'0'*
- Ein Array ohne Elemente.
- Der spezielle Datentyp *NULL*

- Array
  - Ein Array ist ein Datenfeld, das andere Variablen, auch andere Arrays, enthalten kann.
  - Einzelne Elemente in einem PHP-Array können entweder über einen numerischen Index oder einen Suchschlüssel angesprochen werden. Ein solcher Suchschlüssel wird *key* genannt.

  - Ein Array wird zumeist über Schleifen verwaltet. Üblich sind dabei Anweisungen zum Befüllen, zur Suche und zur Ausgabe eines Arrays.
  - Weitere Informationen zu Arrays finden Sie im folgenden Kapitel. Die Syntax von Schleifen in PHP wird in Kapitel 2.1.4 vorgestellt.

- Resource-ID
  - Dabei handelt es sich um einen Zeiger auf eine geöffnete Datenquelle, beispielsweise auf eine Datei oder eine Datenbankverbindung. Solche Zeiger werden auch als Handles bezeichnet.
  - Der Wertebereich ist eine Untermenge des Integer-Wertebereichs.

- Object
  - Eine Objektreferenz ist ein Zeiger auf ein Objekt, das eine Menge von Daten besitzt, die man auch Attribute oder Eigenschaften nennt. Zusätzlich besitzt ein Objekt eine Menge von eigenen Funktionen.
  - Die Möglichkeiten von Objekten wurden in PHP5 stark erweitert. Die theoretischen Grundlagen der Objektorientierung werden im dritten Kapitel, die Umsetzung in PHP im vierten Kapitel besprochen.

- NULL
  - Seit PHP4 gibt es den speziellen Datentyp *NULL*, den man am Besten mit „nichts“ übersetzen kann.
  - Eine Variable ist dann *NULL*, wenn ihr noch kein Wert zugewiesen wurde, wenn sie gelöscht wurde oder wenn ihr direkt *NULL* zugewiesen wurde, z. B. *$x=NULL*.
  - Bei einer Zuweisung oder einer Abfrage auf *NULL* dürfen keine Anführungszeichen verwendet werden, da PHP sonst eine Zeichenkette *“NULL“* erzeugt.

Ausgabe von Variablen

Bei der Ausgabe der Variablen *$g* und *$h* in Listing 2.4 ist für einen Programmierer ungewöhnlich, dass gar keine Ausgabe erscheint. Aus anderen Sprachen hätte man ein *FALSE* oder zumindest 0 erwartet. Die einfache Ausgabe einer Variablen durch *<?php echo $g?>* oder *<?php echo($g); ?>* kann für einen Programmierer irreführend sein und beim Debugging sogar zu falschen Annahmen führen.

> **Profitipp**
>
> Ein erfahrener Programmierer verwendet statt einer direkten Ausgabe eines Variableninhalts zum Debugging den PHP-Befehl *var_dump*. Dieser Befehl gibt sowohl den Datentyp als auch den Inhalt der Variablen zurück.

So erhält man mit *<?php echo var_dump($g)?>* in Listing 2.4 die erwartete Ausgabe im Internetbrowser mit *bool(FALSE)*. Wenn Sie nur der Datentyp einer Variablen interessiert, so können Sie den Befehl *<?php echo gettype($g)?>* verwenden. Im Beispiel wird dann *boolean* zurückgegeben.

Wie bereits erwähnt, ist PHP eine untypisierte Sprache. Für erfahrene Programmierer einer anderen Sprache ist oft auch das Verhalten einer PHP-Variablen im Laufe ihrer

Existenz überraschend. Man kann denken, dass eine Variable bei ihrer ersten Wertzuweisung einen Datentyp erhält und dieser Datentyp dann für die restliche Gültigkeit der Variablen gleich bleibt. Das ist bei PHP jedoch nicht der Fall, wie der Quellcode in Listing 2.6 zeigt.

In diesem Listing wird auch gezeigt, dass PHP-Anweisungen mit einem Semikolon abgeschlossen werden. Das ist immer dann notwendig, wenn mehrere Anweisungen in einem PHP-Block ausgeführt werden. Eine einzelne Anweisung kann auch ohne Semikolon in einem PHP-Skript verwendet werden:

```
<html><body>
  <?php
    $a=4;
    echo(var_dump($a));
    echo('<br>');
    $a=9.9;
    echo(var_dump($a));
    echo('<br>');
    $a=FALSE;
    echo(var_dump($a));
    echo('<br>');
  ?>
</body></html>
```

Listing 2.6: Eine Variable mit veränderlichem Datentyp

Die erste Überraschung besteht darin, dass dieser Quellcode überhaupt funktioniert und dann noch gültige Ausgaben erzeugt, nämlich *int(4)*, *float(9.9)* und in der letzten Zeile *bool(FALSE)*. Eine Variable passt in PHP also bei jeder Wertzuweisung ihren Datentyp dynamisch an.

Das hat auch zur Folge, dass man zu einem Zeitpunkt nicht genau sagen kann, welchen Datentyp eine Variable gerade besitzt, wenn ein anderer Entwickler schreibenden Zugriff auf diese Variable hat.

Abfrage von Variablen und Datentypen

Während *var_dump* nur für die Ausgabe bestimmt ist, hat man mit weiteren Befehlen die Möglichkeit, den Datentyp einer Variablen innerhalb des PHP-Skripts abzufragen. Diese Befehle geben stets einen Wahrheitswert zurück, der dann über eine Verzweigung (Kapitel 2.1.3) Einfluss auf den weiteren Ablauf des PHP-Skripts nehmen kann.

Befehl	Bedeutung
is_int($var) oder is_integer($var) oder is_long($var)	Gibt *TRUE* zurück, wenn *$var* eine Ganzzahl ist.
is_float($var) oder is_double($var) oder is_real($var)	Gibt *TRUE* zurück, wenn *$var* eine Fließkommazahl ist.
is_numeric($var)	Gibt *TRUE* zurück, wenn *$var* eine Zahl ist oder eine Zeichenkette, die man in eine Zahl umwandeln kann.
is_bool($var)	Gibt *TRUE* zurück, wenn *$var* ein Wahrheitswert ist.
is_string($var)	Gibt *TRUE* zurück, wenn *$var* eine Zeichenkette ist.
is_array($var)	Gibt *TRUE* zurück, wenn *$var* ein Datenfeld ist.
is_object($var)	Gibt *TRUE* zurück, wenn *$var* eine Referenz auf ein Objekt ist.

Tabelle 2.1: Befehle zur Prüfung von Datentypen

Um zu prüfen, ob eine Variable *$var* überhaupt existiert, bietet PHP den Befehl *isset($var)* an. So liefert der Quellcode in Listing 2.7 die Ausgabe *bool(TRUE)* und in der nächsten Zeile *bool(FALSE)*:

```
<html><body>
  <?php
    $a=4;
    echo(var_dump(isset($a))); echo('<br>'); echo(var_dump(isset($A)));
  ?>
</body></html>
```

Listing 2.7: Prüfung mit isset

Während *isset($var)* prüft, ob eine Variable *$var* existiert, prüft der Befehl *empty($var)*, ob eine Variable „leer“ ist. Das ist je nach Datentyp unterschiedlich definiert. Bei folgenden Gegebenheiten liefert *empty($var)* den Wert *TRUE* zurück:

- Eine Zeichenkette *$a* ist leer: *$a=““*.
- Eine Zeichenkette *$a* enthält den Wert 0: *$a=“0“*.
- Ein Wahrheitswert ist *FALSE*.
- Eine Variable ist *NULL*.
- Eine Zahl ist 0.
- Ein Datenfeld/Array enthält keine Elemente.

Eine einmal definierte Variable *$var* ist normalerweise im gesamten PHP-Skript gültig. Wenn Sie jedoch eine Variable nicht mehr benötigen und einen weiteren Zugriff auf deren Inhalt verhindern wollen, so können Sie die Variable mit *unset($var)* löschen. Sie erhält dann den Datentyp bzw. den Wert *NULL*. Das Beispiel in Listing 2.8 erzeugt als Ausgabe in der ersten Zeile *float(4.8)* gefolgt von *NULL*:

```
<html><body>
  <?php
    $a=4.8; echo(var_dump($a)); echo('<br>');
    unset($a); echo(var_dump($a));
  ?>
</body></html>
```

Listing 2.8: Löschen einer Variablen mit unset

Umwandlung von Datentypen

Oft kommt es vor, dass ein Anwender über ein Textfeld in einem HTML-Formular Zahlen eingeben muss. Da ein Textfeld jedoch beliebige Eingaben zulässt, werden die Eingaben zunächst als Zeichenkette gespeichert. Mit den Funktionen aus Tabelle 2.1 kann man nun mit einer zusätzlichen Verzweigung prüfen, ob eine Variable in eine Zahl umgewandelt werden kann. Wenn das machbar ist, fehlt noch eine PHP-Funktion, die die Umwandlung wirklich vornimmt. Man spricht hier von einem „Casting" in einen anderen Datentyp. Der Quellcode in Listing 2.9 testet das Casting.

```
<html><body>
  <?php
    $x="3.8";
    $a=(int)$x; $b=(double)$x; $c=(string)$b; $d=(bool)$b;
    echo(var_dump($a)); echo('<br>');
    echo(var_dump($b)); echo('<br>');
    echo(var_dump($c)); echo('<br>');
    echo(var_dump($d)); echo('<br>');
  ?>
</body></html>
```

Listing 2.9: Umwandeln von Datentypen

Die Eingabe *$x* ist eine Zeichenkette, die weiterverarbeitet werden soll. Zunächst erfolgt ein Cast in eine Ganzzahl in der Variablen *$a*, die 3 ergibt. Beim Cast in eine Ganzzahl wird also nicht gerundet, sondern die Nachkommastellen abgeschnitten. Die Umwandlung in eine Fließkommazahl mit *(double)* funktioniert problemlos. Mit dem Wert in der Variablen *$b* kann man nun weiter rechnen. Ebenso problemlos ist die Rück-Wandlung in eine Zeichenkette.

Etwas überraschend ist das Ergebnis bei der Umwandlung der Zeichenkette *"3.8"* in einen Wahrheitswert. Hier wird *TRUE* in der Variablen *$d* gespeichert. Die Ursache liegt darin, dass PHP die Zeichenkette zunächst in eine Zahl umwandelt, die ungleich 0 ist. Die Zahl 0 würde den Wahrheitswert *FALSE* ergeben. Alle anderen Zahlen ergeben *TRUE*.

Wenn Sie den Datentyp einer Variable ändern möchten, ohne einen anderen Variablen-Namen zu verwenden, so können Sie den Befehl *settype($x,"double")* verwenden. Dieser Befehl hat dieselbe Bedeutung wie *$x=(double)$x*, ist jedoch lesbarer.

Verwaltung von Variablen und Referenzen/Zeigern

PHP bietet eine weitere, ungewöhnliche Funktion zur Verwaltung von Variablen. Mit PHP können Sie nicht nur den Wert einer Variablen verändern, sondern auch ihre Bezeichnung. Betrachten Sie dazu das Listing 2.10.

```
<html><body>
  <?php
    $x="name";
    $$x="daten";
    echo($x);
    echo('<br>');
    echo($name);
  ?>
</body></html>
```

Listing 2.10: Variablen in Variablen

Sie definieren zunächst eine Variable *$x*, die mit der Ausgabe *echo($x)* deren Inhalt *"name"* ausgibt. Diesen Inhalt der Variablen benutzen Sie nun mit *$$x="daten"* als Bezeichnung für eine weitere Variable, die den Inhalt *"daten"* erhält. Damit haben Sie eine neue Variable *$name* erstellt, die mit *echo($name)* deren Inhalt, nämlich *"daten"*, zurück gibt. Diesen Inhalt können Sie auch direkt über *echo($$x)* zugreifen.

Bei der Verwaltung von Variablen verhält sich PHP sehr ähnlich wie die Sprache C, die als Vorbild von PHP gilt. Ebenso wie in C können auch in PHP Referenzen auf Daten verwaltet werden, die man auch als Zeiger bezeichnet. Mit *$wert=4* erzeugt man einen Zeiger *$wert*, der auf eine Speicherstelle zeigt, in der ein Integer-Wert, in diesem Fall 4, gespeichert ist. In PHP können mehrere Zeiger auf dieselbe Speicherstelle zeigen. Einen neuen Zeiger *$wert2* erzeugen Sie mit der Anweisung *$wert2=&$wert*. Beachten Sie das kaufmännische „&" vor dem Dollar-Symbol. Wenn Sie nun mit *echo($wert2)* den Wert auslesen, so ist dieser mit der Ausgabe von *$wert* identisch. Interessant wird es dann, wenn Sie *$wert2* ändern, z. B. durch die Anweisung *$wert2=99*. Wenn Sie nun *echo($wert)* bzw. *echo($wer2)* ausführen, stellen Sie fest, dass sowohl *$wert* als auch *$wert2* die Ausgabe 99 liefern:

```
<html><body>
  <?php
    $wert=4;
    echo($wert); echo('<br>');
    $wert2=&$wert;
    echo($wert2); echo('<br>');
    $wert2=99;
    echo($wert);echo('<br>');
    echo($wert2);echo('<br>');
  ?>
</body></html>
```

Listing 2.11: Zwei Variablen zeigen auf die selbe Speicherstelle

Im Gegensatz zu der Referenzübergabe *$wert2=&$wert* kopiert PHP bei der Wertzuweisung *$wert2=$wert* (ohne das kaufmännische „Und“) den Inhalt von *$wert* auf eine neue Speicherstelle, auf die dann *$wert2* zeigt. Wenn Sie dann *$wert2* ändern, hat das keinen Einfluss auf die Speicherstelle, auf die *$wert* zeigt. Abbildung 2.4 verdeutlicht diesen Sachverhalt.

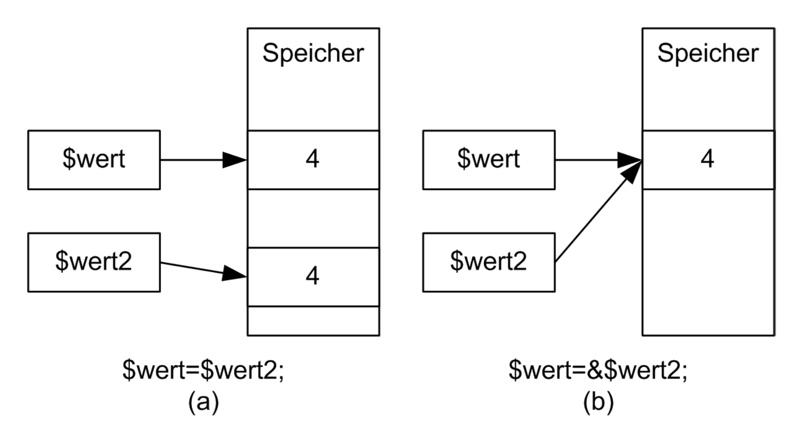


Abbildung 2.4: Situation im Speicher bei einer Wert- (a) und einer Referenzübergabe (b)

Operatoren

In den bisherigen Beispielen wurden bereits Zuweisungen von Werten zu Variablennamen vorgenommen. Im letzten Kapitel wurde zwischen der Zuweisung von Werten und Speicherreferenzen unterschieden. Tabelle 2.2 fasst die möglichen Operatoren von PHP zusammen, mit denen man Zuweisungen erstellen kann.

Zusätzlich dazu existieren einige Operatoren für mathematische Grundrechenarten. Komplexere mathematische Funktionen wie Sinus- oder Logarithmusfunktionen werden in Kapitel 2.1.5 beschrieben.

Operation	Bedeutung
$x = $y	Inhalt der Variablen *$y* wird nach *$x* kopiert
$x = &$y	Speicheradresse der Variablen *$y* wird auf die Speicheradresse der Variablen *$x* gesetzt
$x += $y	Addition von *$y* zur Variablen *$x*
$x -= $y	Substraktion von *$y* von der Variablen *$x*
$x *= $y	Multiplikation von *$y* mit der Variablen *$x*
$x /= $y	Division von *$x* mit *$y*
$x %= $y	Rest der Ganzzahldivision von *$x* und *$y*
$x .= $y	Hinzufügen der Zeichenkette *$y* zu der Zeichenkette *$x*

Tabelle 2.2: Kombinierte Zuweisungsoperatoren

Ebenso können Sie die Ergebnisse von mathematischen Berechnungen in separaten Variablen ablegen. Interessant in beiden Fällen ist der Punktoperator zum Verbinden von Zeichenketten, da eine große Aufgabe von PHP in der Verarbeitung von Zeichenketten besteht. In diesen Zeichenketten werden Teile von HTML-, JavaScript- und/oder CSS-Code verwaltet, die dann zum Client gesendet werden. Dort werden sie in einem Internetbrowser interpretiert.

Operation	Bedeutung
$z = $x + $y	Addition von *$x* und *$y* in die Variable *$z*
$z = $x - $y	Subtraktion von *$x* und *$y* und Speicherung in die Variable *$z*
$z = $x * $y	Multiplikation von *$x* und *$y* und Speicherung in die Variable *$z*
$z = $x / $y	Division von *$x* mit *$y* und Speicherung in die Variable *$z*
$z = $x % $y	Rest der Ganzzahldivision von *$x* und *$y* und Speicherung in die Variable *$z*
$z = $x . $y	Aneinanderreihen der Zeichenketten *$x* und *$y* und Speicherung in die Variable *$z*

Tabelle 2.3: Zuweisungsoperatoren

Zusätzlich existieren einige Operatoren, um Variablen zu vergleichen. Diese Operatoren liefern entweder *TRUE* oder *FALSE* zurück und werden insbesondere bei Verzweigungen (Kapitel 2.1.3) und Schleifen (Kapitel 2.1.4) eingesetzt.

Operation	Bedeutung
$x > $y	liefert *TRUE*, wenn der Wert von *$x* größer ist als der Wert von *$y*
$x >= &$y	liefert *TRUE*, wenn der Wert von *$x* größer oder gleich dem Wert von *$y* ist
$x < $y	liefert *TRUE*, wenn der Wert von *$x* kleiner ist als der Wert von *$y*
$x <= $y	liefert *TRUE*, wenn der Wert von *$x* kleiner oder gleich dem Wert von *$y* ist
$x == $y	liefert *TRUE*, wenn der Wert von *$x* gleich dem Wert von *$y* ist
$x != $y	liefert *TRUE*, wenn der Wert von *$x* ungleich dem Wert von *$y* ist
$x === $y	liefert *TRUE*, wenn der Wert von *$x* gleich dem Wert von *$y* ist und *$x* und *$y* vom selben Typ sind
$x !== $y	liefert *TRUE*, wenn der Wert von *$x* ungleich dem Wert von *$y* ist und die Datentypen von *$x* und *$y* unterschiedlich sind

Tabelle 2.4: Vergleichsoperatoren

Die Prüfung auf Gleichheit besteht also aus einem doppelten Gleichheitszeichen, um eine Prüfung von einer Zuweisung zu unterscheiden. Zusätzlich wird noch ein dreifaches Gleichheitszeichen als Operator eingeführt, um auch ein Vergleich von Wert und Datentyp zu ermöglichen. So liefert Listing 2.12 in der ersten Ausgabe *TRUE* und in der zweiten Ausgabe *FALSE*:

```
<html><body>
  <?php
    $x=0;
    $y=0.0;
    echo(var_dump($x==$y)); echo('<br>');
    echo(var_dump($x===$y)); echo('<br>');
  ?>
</body></html>
```

Listing 2.12: Prüfen auf Gleichheit der Werte und/oder Datentypen

Ebenso wie Vergleichsoperatoren liefern auch logische Operatoren Wahrheitswerte als Ergebnis, sodass sie sich für Verzweigungen und Schleifen eignen. Der Sinn von logischen Operatoren liegt in ihrer Verknüpfung zu komplexen Bedingungen, die den Ablauf des Quellcodes beeinflussen. Ein Beispiel ist *$ergebnis = ($x > 20) and (($y < 0) or !$z)*. Tabelle 2.5 liefert eine Übersicht über logische Operatoren.

<table>
<tr><th>Operation</th><th>Bedeutung</th></tr>
<tr><td>$x and $y bzw.
$x && $y</td><td>liefert TRUE, wenn sowohl der Wert/Ausdruck von $x als auch der Wert/Ausdruck von $y TRUE ist</td></tr>
<tr><td>$x or $y bzw.
$x || $y</td><td>liefert TRUE, wenn sowohl der Wert/Ausdruck von $x als auch der Wert/Ausdruck von $y TRUE ist oder nur einer der beiden Werte/Ausdrücke TRUE ist</td></tr>
<tr><td>$x xor $y</td><td>liefert TRUE, wenn der Wert/Ausdruck von $x oder der von $y TRUE ist, aber nicht beide</td></tr>
<tr><td>! $x</td><td>liefert TRUE, wenn der Wert/Ausdruck von $x FALSE ist</td></tr>
</table>

Tabelle 2.5: Logische Operatoren

Wie andere Programmiersprachen bietet auch PHP den Zugriff auf einzelne Bits von Variablen.

<table>
<tr><th>Operation</th><th>Bedeutung</th></tr>
<tr><td>$x & $y</td><td>binäre UND-Verknüpfung</td></tr>
<tr><td>$x | &$y</td><td>binäre ODER-Verknüpfung</td></tr>
<tr><td>$x ^ $y</td><td>binäre XOR-Verknüpfung (liefert TRUE, wenn $x oder $y wahr ist, aber nicht beide)</td></tr>
<tr><td>~ $x</td><td>Komplement-Darstellung</td></tr>
<tr><td>$x << $y</td><td>verschiebt die Bits von $a um $b-Schritte nach links; jeder Schritt nach links bedeutet eine Multiplikation mit 2</td></tr>
<tr><td>$x >> $y</td><td>verschiebt die Bits von $a um $b-Schritte nach rechts; jeder Schritt nach rechts bedeutet eine Division durch 2</td></tr>
</table>

Tabelle 2.6: Bit-Operatoren

Abschließend werden noch die Inkrement- und Dekrementoperatoren vorgestellt, wie sie auch in Sprachen wie C und Java verwendet werden. In einer Vielzahl von Anwen-

dungen ist das Verändern einer Variable um 1 von Bedeutung, beispielsweise um in einem Datenfeld zu suchen. Statt einer Operation mit anschließender Wertzuweisung wie *$x=$x+1* oder *$x+=1* kann in PHP eine übersichtlichere und performantere Art der Schreibweise gewählt werden, nämlich *$x++*.

Operation	Bedeutung
++$x	erhöht den Wert von *$x* um 1 und gibt den neuen Wert von *$x* zurück
$x++	gibt den Wert von *$x* zurück und erhöht den Wert anschließend um 1
--$x	verringert den Wert von *$x* um 1 und gibt den neuen Wert von *$x* zurück
$x--	gibt den Wert von *$x* zurück und vermindert den Wert anschließend um 1

Tabelle 2.7: Inkrement- und Dekrementoperatoren

2.1.2 Datenfelder: Arrays

Es wurde bereits kurz erwähnt, dass es sich bei Arrays um Datenfelder handelt, also um zusammengefasste Sammlungen von Variablen. Die Verwaltung von Arrays erfolgt in PHP sehr dynamisch, sodass flexible Datenstrukturen erzeugt werden können. Einerseits muss die Größe eines Datenfeldes nicht im Vorfeld bekannt sein und andererseits kann ein Datenfeld aus Variablen verschiedener Datentypen bestehen. Ein Array wird über einen Index angesprochen, dessen Wert normalerweise mit 0 beginnt.

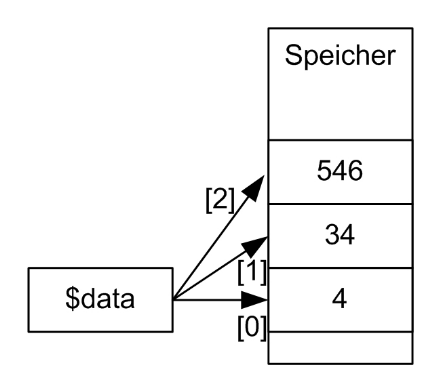


Abbildung 2.5: Zugriff auf ein Array

Das in Abbildung 2.5 dargestellte Array kann mit dem Quellcode aus Listing 2.13 erzeugt werden. Der Zugriff erfolgt über den Bezeichner des Arrays mit anschließender Angabe des Index. Da der Index aus Zahlen besteht, spricht man von einem numerischen Array:

```
<html><body>
  <?php
    $data=Array();$data[]=4;$data[]=34;$data[]=546;
    echo(var_dump($data[0])); echo('<br>');
```

Listing 2.13: Das erste numerische Array

```
    echo(var_dump($data[1])); echo('<br>');
    echo(var_dump($data[2])); echo('<br>');
    echo(var_dump($data)); echo('<br>');
  ?>
</body></html>
```

Listing 2.13: Das erste numerische Array (Forts.)

Mit *var_dump* können Sie die Anzahl der einzelnen Elemente jeweils mit ihrem Datentyp und Wert ausgeben. Ebenso können Sie Operationen mit den Elementen durchführen, z.B. *$istgrößer3=($data[0]>3)*.

> **Profitipp**
>
> Wenn Sie *var_dump* auf das Array selbst anwenden, also *var_dump($data)*, so erhalten Sie eine detaillierte Auflistung der Inhalte des Arrays, nämlich:
>
> *array(3) { [0]=> int(4) [1]=> int(34) [2]=> int(546) }.*

Eine alternative Erzeugung des Arrays kann in einem Schritt über den Befehl *$data=array(4, 34, 546);* erfolgen.

Beliebige Startwerte und negative, nicht fortlaufende Indizes

Den Startwert des Arrays können Sie auch frei wählen, indem Sie ihn bei der Erzeugung des Arrays angeben, beispielsweise *$kollegen=array(10 => "Uli", "Hanz", "Kurt");*. Die Indizes lauten dann 10, 11 und 12.

Ebenso sind im Gegensatz zu anderen Programmiersprachen auch negative Indizes möglich. Dann muss der Index jedoch bei jedem Element angegeben werden, da ansonsten eine neue Indizierung bei 0 beginnt, also *$kollegen=array(-3 => "Uli", -4 => "Hans", -5 => "Kurt");*.

Bei *$kollegen=array(-3 => "Uli", "Hans", "Kurt");* ergibt *echo(var_dump($kollegen));* die Ausgabe *array(3) { [-3]=> string(5) "Frank" [0]=> string(4) "Kurt" [1]=> string(4) "Hans" }*. Hier erkennen Sie bereits, dass die Indizierung nicht fortlaufend sein muss. Sie können für den Index beliebige Werte wählen.

Bezeichner statt Indizes: Assoziative Felder

Die Indizierung muss nicht einmal aus numerischen Werten bestehen, sondern lediglich aus eindeutigen Bezeichnern. Das ist in Listing 2.14 dargestellt. Man spricht hier von einem assoziativen Feld oder einem Hash-Feld. Statt *var_dump* wird in diesem Beispiel die Konkatenation der Zeichenketten mit dem Punktoperator durchgeführt. Außerdem werden die etwas performanteren, einzelnen Anführungszeichen bei allen Zeichenketten verwendet:

```
<html><body>
  <?php
    $personen=array(
    'ich' => 'Frank',
    'Vater' => 'Kurt',
    'Bekannter' => 'Hans');
    echo($personen[ich].'<br>'.$personen[Vater].'<br>');
    echo($personen[Bekannter].'<br>');
  ?>
</body></html>
```

Listing 2.14: Ein Index aus Zeichenketten

Bei ihrer Verwendung müssen die Indizes nicht in einzelnen oder doppelten Anführungszeichen gesetzt werden. Das erhöht die Lesbarkeit des Quellcodes.

Dynamische Inhalte

Wie bereits erwähnt, müssen auch die Datentypen in einem Array nicht einheitlich sein. Das wird in Listing 2.15 verdeutlicht. Somit sind Arrays in PHP sehr dynamische Datenfelder, die mit Collections in Java vergleichbar sind:

```
<html><body>
  <?php
    $data=Array(); $data[2]=0; $data[-5]=0.0; $data[8]="Hallo";
    echo(var_dump($data));
  ?>
</body></html>
```

Listing 2.15: Ein Array mit dynamischen Datentypen und Indizes

Die Ausgabe dieses Arrays lautet *array(3) { [2]=> int(0) [-5]=> float(0) [8]=> string(5) "Hallo" }*.

Löschen von Feldern

PHP bietet Ihnen die Möglichkeit, ganze Arrays zu löschen oder zu leeren. Zusätzlich können Sie einzelne Elemente aus einem Array entfernen. Listing 2.16 zeigt, wie Sie zunächst das Element mit dem Index 8 aus Listing 2.15 löschen, dann das Datenfeld *$data* leeren und abschließend die Referenz *$data* selbst entfernen. Wenn Sie ein ganzes Array nicht mehr benötigen, so können Sie direkt die Referenz entfernen. PHP organisiert dabei das gesamte Speichermanagement:

```
<html><body>
  <?php
    $data=Array(); $data[2]=0; $data[-5]=0.0; $data[8]="Hallo";
```

Listing 2.16: Löschen eines Elements sowie Löschen des gesamten Datenfelds

```
    unset($data[8]); // löschen eines Elementes
    echo(var_dump($data).'<br>');
    $data=Array(); // leeren des Arrays
    echo(var_dump($data).'<br>');
    unset($data); // löschen des Arrays
    echo(var_dump($data).'<br>');
  ?>
</body></html>
```

Listing 2.16: Löschen eines Elements sowie Löschen des gesamten Datenfelds (Forts.)

Die Ausgabe dieses Skripts lautet:

array(2) { [2]=> int(0) [-5]=> float(0) }

array(0) { }

NULL

Mehrdimensionale Datenfelder

Genauso dynamisch, wie Sie Elemente in einem Feld verwalten können, können Sie auch Felder in einem Feld verwalten. Das führt zu mehrdimensionalen Arrays. Eine Möglichkeit, ein mehrdimensionales Array anzulegen, zeigt Listing 2.17. Dort ist auch der Zugriff auf das mehrdimensionale Datenfeld dargestellt. Die Ausgabe ist identisch mit der Reihenfolge, in dem die einzelnen Elemente angelegt worden sind:

```
<html><body>
  <?php
    $kunden=array(
      array('Frank','Dopatka'),
      array('Uli','Müller'),
      array('Max','Mustermann')
    );
    echo($kunden[0][0].' '.$kunden[0][1].'<br>');
    echo($kunden[1][0].' '.$kunden[1][1].'<br>');
    echo($kunden[2][0].' '.$kunden[2][1].'<br>');
  ?>
</body></html>
```

Listing 2.17: Erstes mehrdimensionales Datenfeld

Ebenso können Sie auch bei einem mehrdimensionalen Datenfeld die Indizes frei wählen. Das Beispiel in Listing 2.18 zeigt eine freie Definition der Indizes:

```
<html><body>
  <?php
    $kunden=array(
      'K1'=>array('VN'=>'Frank','N'=>'Dopatka'),
      'K2'=>array('VN'=>'Uli','N'=>'Müller'),
      'K3'=>array('VN'=>'Max','N'=>'Mustermann')
    );
    echo($kunden[K1][VN].' '.$kunden[K1][N].'<br>');
    echo($kunden[K2][VN].' '.$kunden[K2][N].'<br>');
    echo($kunden[K3][VN].' '.$kunden[K3][N].'<br>');
  ?>
</body></html>
```

Listing 2.18: Ein mehrdimensionales Datenfeld mit eigenen Indizes

Wie auch bei einem eindimensionalen Array muss ein mehrdimensionales Array nicht bei seiner Initialisierung mit Werten gefüllt werden. Die zweite Dimensionierung kann ebenso dynamisch durchgeführt werden, wie Listing 2.19 zeigt:

```
<html><body>
  <?php
    $kunden=Array();
    $kunden[0]=Array();
    $kunden[0][0]='Frank';
    $kunden[0][1]='Dopatka';
    $kunden[1]=Array();
    $kunden[22][0]='Uli';
    $kunden[22][1]='Müller';
    echo($kunden[0][0].' '.$kunden[0][1].'<br>');
    echo($kunden[22][0].' '.$kunden[22][1].'<br>');
  ?>
</body></html>
```

Listing 2.19: Ein dynamisch initialisiertes mehrdimensionales Datenfeld

PHP-Funktionen zur Bearbeitung von Datenfeldern

Die Sprache PHP verfügt über mächtige Funktionen zur Bearbeitung von Datenfeldern. Die Wichtigsten dieser Funktionen sind im Folgenden kurz zusammengefasst. Wenn Sie eine vollständige Übersicht erhalten möchten, können Sie in Suchmaschinen wie Google mit Begriffen wie „PHP", „Array" und „Funktionen" weitere, seltener verwendete Funktionen abrufen. Beispielsweise hat sich die Internetseite *http://www.phpcenter.de/de-html-manual/ref.array.html* als sehr übersichtlich herausgestellt.

Für Sie als Programmierer ist es sinnvoll zu wissen, welche Funktionen PHP bereits bereit stellt, damit Sie bei Bedarf darauf zugreifen können. Es ist nicht ratsam, dass Sie

existierende Funktionen nachprogrammieren, da Sie mit Sicherheit weder die Performance, noch die Fehlerfreiheit von PHP erreichen.

Die erste Gruppe von Funktionen umfasst das Sortieren von Feldern. Neben einer zufälligen Anordnung können Sie eine auf- bzw. absteigende Sortierung wählen.

Funktion	Bedeutung
shuffle($arr)	ordnet alle Elemente des Datenfeldes zufällig neu an
sort($arr)	sortiert ein eindimensionales Datenfeld vorwärts; war es ein assoziatives Feld, so wird es in ein nummerisches Feld umgewandelt
rsort($arr)	sortiert ein eindimensionales Datenfeld rückwärts; war es ein assoziatives Feld, so wird es in ein nummerisches Feld umgewandelt
asort($arr)	sortiert ein eindimensionales Datenfeld vorwärts und behält die Beziehungen in einem assoziativen Feld bei
arsort($arr)	sortiert ein eindimensionales Datenfeld rückwärts und behält die Beziehungen in einem assoziativen Feld bei

Tabelle 2.8: Sortierung eines Felds

Listing 2.20 zeigt einen Anwendungsfall einer Sortierung. Die Ausgabe lautet „Frank“, „Max“ und dann „Uli“. Auf diese Weise können Sie mit eigenen, einfachen Beispielen jede Funktion austesten:

```
<html><body>
  <?php
    $kunden=array('Frank','Uli','Max');
    sort($kunden);
    echo(var_dump($kunden));
  ?>
</body></html>
```

Listing 2.20: Sortieren eines Datenfelds

Zusätzlich können Sie auch eine Sortierung nach einer eigenen Funktion durchführen. Wie Sie eine Funktion deklarieren und damit ein Datenfeld sortieren, wird in Kapitel 2.1.5 vorgestellt.

Funktion	Bedeutung
usort($arr, func)	sortiert ein eindimensionales Datenfeld nach einer eigenen Funktion
uasort($arr, func)	sortiert ein assoziatives Datenfeld nach einer eigenen Funktion

Tabelle 2.9: Funktionen zur Feldsortierung mit eigener Funktion

Ähnlich wie beim Auslesen einer Ergebnismenge aus einer Datenbank (Kapitel 2.2: „Zugriff auf eine MySQL-Datenbank“) kann man auch einen Zeiger über ein Datenfeld laufen lassen. Das wird oft bei der Suche nach einzelnen Elementen oder für eine Weiter-

verarbeitung verwendet. Zusätzlich können Sie ein Datenfeld nach der Anzahl der enthaltenen Elemente fragen.

Funktion	Bedeutung
count($arr) oder size_of($arr)	gibt die Anzahl der Elemente im Datenfeld zurück
reset($arr)	setzt den internen Zeiger im Datenfeld auf das erste Element
end($arr)	setzt den internen Zeiger im Datenfeld auf das letzte Element
current($arr) oder pos($arr)	gibt den Inhalt des Elements zurück, auf dem der Zeiger steht
key($arr)	gibt den Index des Elements zurück, auf dem der Zeiger steht
next($arr)	setzt den internen Zeiger im Datenfeld um 1 nach vorne
prev($arr)	setzt den internen Zeiger im Datenfeld um 1 zurück
array_walk($arr, func)	wendet eine selbst definierte Funktion auf jedes Element des Datenfelds an

Tabelle 2.10: Zugriffsfunktionen auf ein Datenfeld

Interessant ist auch die Funktion *array_walk*, die eine eigene Funktion[2] auf jedes Element im Feld anwendet. Damit können Sie eine Schleifenstruktur[3] einsparen und erzeugen übersichtlichen Quellcode.

In Listing 2.21 wird der Zeiger auf ein Datenfeld *$kunden* zunächst auf das erste Element mit der *reset*-Funktion gesetzt. Dieses Element wird dann über die *key*- und *current*-Funktion ausgegeben. Im Anschluss daran wird der Zeiger über die *next*-Funktion weiter mit erneuter Ausgabe bewegt:

```
<html><body>
  <?php
    $kunden=array('Frank','Uli','Max');
    reset($kunden);
    echo('Position:'.key($kunden).'<br>');
    echo('Wert:'.current($kunden).'<br>');
    next($kunden);
    echo('Position:'.key($kunden).'<br>');
    echo('Wert:'.current($kunden).'<br>');
  ?>
</body></html>
```

Listing 2.21: Durchlaufen eines Datenfelds

2 Kapitel 2.1.5
3 Kapitel 2.1.4

Abschließend werden weitere Funktionen zur Behandlung von Datenfeldern vorgestellt, die sich im Alltag als sinnvoll erweisen. Diese betreffen unter anderem die Verwaltung mehrerer Felder.

Ebenso ist die Funktion *in_array* hervorzuheben, die eine automatische Suche nach einem Element durchführt. Aus eigener Erfahrung wird diese Funktion selten verwendet und stattdessen eine kompliziertere und inperformante Schleifenkonstruktion.

Funktion	Bedeutung
array_diff($arr1, $arr2,...)	ermittelt Unterschiede in Datenfeldern und gibt diese als neues Datenfeld zurück
array_merge($arr1, $arr2)	verbindet zwei Datenfelder zu einem neuen Feld
array_pad($arr, $len, $wert)	verkürzt (bei *$len<0*) oder verlängert ein nummerisches Feld um *$len* Elemente und ersetzt leere Elemente
in_array($wert, $arr)	gibt *TRUE* zurück, wenn ein Wert in einem Datenfeld vorhanden ist
array_shift($arr)	liefert den Wert des ersten Elements eines Datenfelds und löscht das Element dann im Feld
array_pop($arr)	gibt den Wert des ersten Elements eines Datenfelds und löscht das Element dann im Feld
array_sum($arr)	summiert die Werte aller Ganz- und Fließkommazahlen aus einem Feld
array_unique($arr)	entfernt mehrfache Einträge aus einem Datenfeld

Tabelle 2.11: Weitere nützliche Datenfeldfunktionen

2.1.3 Verzweigungen

In diesem Kapitel wird ein wichtiges Konzept jeder prozeduralen Programmiersprache vorgestellt. Mithilfe einer Verzweigung sind Sie in der Lage, eine alternative Ausführung Ihrer Anwendung zu programmieren. Sie können damit auf eine eintretende Bedingung entsprechend reagieren.

Ein Beispiel ist das Anmelden an ein Internetportal. Wenn Sie eine richtige Kombination von Benutzername und Kennwort eingeben, so sollen Sie in das Portal gelangen. Ansonsten soll eine Fehlermeldung angezeigt werden. Die richtigen Angaben soll PHP später aus einer Datenbank entnehmen. Dabei muss eine Verbindung zur Datenbank aufgebaut werden. Nun kann es vorkommen, dass der Datenbankserver nicht online ist. In diesem Fall kann man nicht prüfen, ob die richtigen Daten eingegeben wurden. Wenn die Verbindung also hergestellt wurde, kann erst die Prüfung erfolgen. Ansonsten muss der Anwender eine Fehlermeldung erhalten, dass technische Probleme bei der Prüfung aufgetreten sind, wofür sich der Anbieter entschuldigt.

Immer wenn Sie im Sprachgebrauch eine „Wenn-Dann"- oder „Wenn-Dann-Ansonsten"- Formulierung verwenden, werden Sie in Ihrer Anwendung eine Verzweigung einsetzen. Eine Verzweigung ist auch immer an eine Bedingung geknüpft, die entweder erfüllt oder nicht erfüllt sein kann. Bei der Bedingung handelt es sich demnach um einen

Wahrheitswert, der *TRUE* oder *FALSE* sein kann. Eine Bedingung kann dabei aus Teilbedingungen bestehen, die über logische Operatoren verknüpft werden (Tabelle 2.5).

Die If-then-else-Verzweigung

PHP bietet wie nahezu alle anderen Programmiersprachen auch die *if-then*-Verzweigung mit der in Listing 2.22 skizzierten Syntax:

```
<?php
  echo('Anweisung1<br>'); echo('Anweisung2<br>');
  if ($bedingung==TRUE){
    echo('Anweisung 3(Bedingung erfüllt)<br>');
  }
  echo('Anweisung4<br>'); echo('Anweisung5<br>'); echo('Anweisung6<br>');
  ?>
```

Listing 2.22: Skizze einer Wenn-Dann-Verzweigung

Zunächst werden die Texte *Anweisung1* und *Anweisung2* ausgegeben. Im weiteren Verlauf werden statt der einfachen Textausgaben PHP-Befehle und/oder eigene Funktionen ausgeführt. In diesem Schritt wird sich jedoch auf die Verzweigung konzentriert. Der Text *Anweisung3(Bedingung erfüllt)
* wird nur dann ausgegeben, wenn die Variable *$bedingung* den Wert *TRUE* hat. Wie es bei PHP typisch ist, wird die Ausgabe zum Internetbrowser des Clients weitergegeben, der dann den Zeilenumbruch interpretiert. Im Anschluss daran werden die Ausgaben *Anweisung4*, *Anweisung5* und *Anweisung6* getätigt. War der Wert der Bedingung *FALSE*, so wird also die Ausgabe *Anweisung3* unterbunden.

Innerhalb der *if*-Konstruktion – also innerhalb der geschweiften Klammern – können beliebig viele Befehle platziert werden, die wie gewohnt durch ein Semikolon voneinander getrennt werden. Außerdem können weitere Verzweigungen platziert werden. So können Sie erreichen, dass beispielsweise zunächst geprüft wird, ob eine Verbindung zur Datenbank erfolgreich aufgebaut wurde. Ist das der Fall, wird das richtige Kennwort aus der Datenbank geholt und mit dem eingegebenen Kennwort verglichen. Dieser Vergleich liefert wiederum einen Wahrheitswert. Wenn dieser *TRUE* ist, hat der Anwender Zugang zum Portal.

Für den Fall, dass eine Bedingung *FALSE* ergibt, können Sie ebenso Anweisungen platzieren, die ausschließlich in diesem Fall ausgeführt werden. Eine Skizze dazu sehen Sie in Listing 2.23:

```
<?php
  echo('Anweisung1<br>'); echo('Anweisung2<br>');
  if ($bedingung==TRUE){
    echo('Anweisung 3(Bedingung erfüllt)<br>');
  }
```

Listing 2.23: Skizze einer Wenn-Dann-Ansonsten-Verzweigung

```
  else{
    echo('Anweisung 4(Bedingung NICHT erfüllt)<br>');
  }
  echo('Anweisung5<br>'); echo('Anweisung6<br>');
  ?>
```

Listing 2.23: Skizze einer Wenn-Dann-Ansonsten-Verzweigung (Forts.)

Hier wird entweder *Anweisung3* ausgegeben, in dem Fall, dass *$bedingung* erfüllt ist, oder *Anweisung4*, falls die *$bedingung* nicht erfüllt ist. Ansonsten wird der Quellcode wie gewohnt linear abgearbeitet.

Abbildung 2.6 stellt die beiden bislang vorgestellten Arten der Verzweigung anhand von Programmablaufplänen gegenüber und bezieht sich dabei auf die Quellcodes der beiden Listings. Um den Verlauf eines Programms darzustellen, eignen sich Programmablaufpläne (PAP) besonders gut. Sie sind nach DIN 66001 genormt. Die Aktivitätsdiagramme der UML (Kapitel 3) können als Weiterentwicklung der Programmablaufpläne gesehen werden.

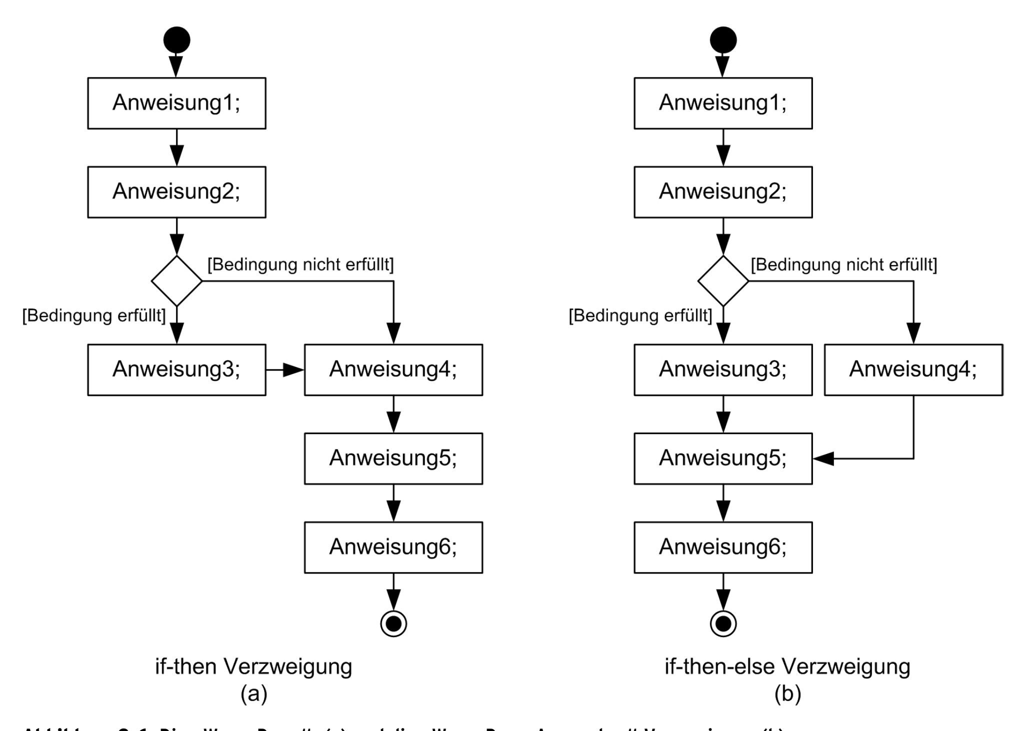


Abbildung 2.6: Die „Wenn-Dann"- (a) und die „Wenn-Dann-Ansonsten"-Verzweigung (b)

Im Folgenden werden einige Szenarien beschrieben, bei denen Verzweigungen im Kontext der HTML-Ausgabe häufig eingesetzt werden. Falls Sie bislang noch nicht mit PHP programmiert haben, sollten Sie diese kleinen Beispiele nachprogrammieren, selbst verändern und verstehen.

Listing 2.24 zeigt bereits eine verschachtelte Verzweigung, bei der eine Variable *$wert* auf eine Grenze, im Beispiel 10, geprüft wird. Es gibt hier drei Möglichkeiten: Die Grenze kann unterschritten oder überschritten sein. Ist beides nicht der Fall, so entspricht der Wert genau dem Grenzwert. In diesem Beispiel wird die Ausgabe über den *echo*-Befehl von PHP in den HTML-Ausgabestrom geschrieben:

```
<?php $wert=10; ?>
<html><body>
  Der Wert ist
  <?php
    if($wert<10){
      echo(' kleiner als ');
    }
    else{
      if($wert>10){
        echo(' größer als ');
      }
    else{
      echo(' gleich ');
    }
  }
?>
10.
</body></html>
```

Listing 2.24: Prüfung eines Werts

PHP bietet Ihnen aber auch die Option, den PHP-Quellcode beim Eintreffen einer Bedingung zu unterbrechen und direkt mit der HTML-Ausgabe fortzufahren. Damit können Sie ununterbrochenen HTML-Code schreiben. Listing 2.25 zeigt das gleiche Beispiel mit direkter HTML-Ausgabe, indem mit *?>HTML<?php* der PHP-Code unterbrochen wird. Dabei werden auch einige HTML-Befehle in die Ausgabe integriert. Welche Art der Ausgabe Sie wählen, liegt an Ihrem eigenen Ermessen und am eigenen Programmierstil:

```
<?php $wert=10; ?>
<html><body>
<?php
  if($wert<10){
    ?><h2>Der Wert ist <b>kleiner als</b> 10.</h2><?php
  }
  else{
    if($wert>10){
      ?><h2>Der Wert ist <b>größer als</b> 10.</h2><?php
```

Listing 2.25: Prüfung eines Werts mit unterbrochenem PHP-Code

```
    }
    else{
      ?><h2>Der Wert ist <b>gleich</b> 10.</h2><?php
    }
  }
?>
</body></html>
```

Listing 2.25: Prüfung eines Werts mit unterbrochenem PHP-Code

Im Beispiel des Listings 2.26 wird anhand einer Verzweigung eine Farbe im HTML-Code gesetzt. Damit kann ein positiver Wert grün und ein negativer Wert rot dargestellt werden. Ist der Wert 0, so wird er schwarz hinterlegt. Um die Farbe zu setzen, definiert PHP einen CSS-Stil (Cascading Stylesheets), der dem Wert dann zugewiesen wird. Ein solcher CSS-Stil definiert die grafische Darstellung von HTML-Code und besitzt eine Vielzahl von Gestaltungsmöglichkeiten.

Die Farbe wird in RGB-Anteilen (Rot-Grün-Blau) im hexadezimalen Format angegeben. Die Farbe Rot entspricht dabei „FF0000“, wobei FF im Hex-Format der Zahl 255 im dezimalen Format entspricht. Da die ersten beiden Stellen der Farbangabe die Rot-Angabe darstellt, wird bei „FF0000“ der maximale Rot-Anteil gesetzt. Die Farbe besitzt weder Grün-, noch Blau-Anteile. Somit ergibt sich ein strahlendes Rot als Ausgabe bei einem negativen Wert:

```
<?php
  $wert=-5.3;
  if($wert<0){
    $farbe='#FF0000';
  }
  else{
    if($wert>0){
      $farbe='#00FF00';
    }
    else{
      $farbe='#000000';
    }
  }
?>
<html><body>
  <font face="Arial,Helvetica" color="<?php echo $farbe?>">
  <?php echo $wert?>
  </font>
</body></html>
```

Listing 2.26: Definition einer Farbe für eine HTML-Ausgabe

Achten Sie darauf, dass bei der HTML-Ausgabe das *color*-Attribut des *font*-Befehls in Anführungszeichen gesetzt werden sollte. Dort hinein gibt PHP den aktuellen Farbwert aus, bevor die Ausgabe zum Internetbrowser des Clients gesendet wird.

In Listing 2.27 ist die resultierende HTML-Ausgabe dargestellt. Das Ergebnis ist eine gewöhnliche HTML-Datei mit einigen CSS-Angaben.

```
<html><body>
  <font face="Arial,Helvetica" color="#FF0000">
  -5.3  </font>
</body></html>
```

Listing 2.27: HTML-Ausgabe von Listing 2.26

Die elseif-Verzweigung

Oft kommt es vor, dass Sie mehrere Fallunterscheidungen schachteln müssen. Die *if-then-else*-Konstruktionen sind in diesem Fall unübersichtlich. Deshalb existiert eine weitere, vereinfachte Struktur, die genau für mehrfache Bedingungen geeignet ist. Die *if-elseif*-Verzweigung wird in Listing 2.28 vorgestellt, indem das Beispiel der Farbauswahl von Listing 2.26 umgeschrieben wird. Der ausgegebene HTML-Code des Beispiels bleibt identisch. Bereits bei der Verwendung von zwei Bedingungen wirkt der *elseif*-Befehl kompakter:

```
<?php
  $wert=0;
  if($wert<0){
    $farbe='#FF0000';
  }
  elseif($wert>0){
    $farbe='#00FF00';
  }
  else{
    $farbe='#000000';
  }
?>
```

Listing 2.28: Definition einer Farbe für eine HTML-Ausgabe mit elseif

Die *elseif*-Blöcke können mehrfach hintereinander kopiert werden. Bei der ersten zutreffenden Bedingung wird der Inhalt des entsprechenden Quellcodeblocks abgearbeitet. Nach der Abarbeitung wird der Quellcode hinter der letzten Klammer von *if-elseif* weiter bearbeitet.

Im Beispiel wurde ausschließlich die Variable *$wert* für die Formulierung der Bedingung verwendet. In diesem Fall können Sie stattdessen auch die noch kompaktere Darstellung der *switch*-Anweisung verwenden, die im nächsten Unterkapitel vorgestellt wird.

Über logische Operatoren lassen sich mehrere Variablen zu komplexen Ausdrücken verbinden und als Bedingung für die *elseif*-Blöcke verwenden. Im Beispiel des Listings 2.29

wird eine ungewöhnliche Urlaubsregelung implementiert. Dabei bekommen Frauen stets etwas mehr Urlaub als Männer. Zusätzlich gibt es eine Staffelung in drei Altersklassen:

```
<?php
  $alter=23; $istWeiblich=TRUE;
  if(($alter<20)&&($istWeiblich)){
    $urlaubstage=26; // jünger als 20 und Frau
  }
  elseif(($alter<20)&&(!$istWeiblich)){
    $urlaubstage=25; // jünger als 20 und Mann
  }
  elseif(($alter>=20)&&($alter<40)&&($istWeiblich)){
    $urlaubstage=28; // 20 incl bis 40 excl. und Frau
  }
  elseif(($alter>=20)&&($alter<40)&&(!$istWeiblich)){
    $urlaubstage=26; // 20 incl bis 40 excl. und Mann
  }
  elseif(($alter>=40)&&($istWeiblich)){
    $urlaubstage=32; // 40 oder älter und Frau
  }
  else{
    $urlaubstage=29; // 40 oder älter und Mann
  }
?>
<html><body><?php echo $urlaubstage?></body></html>
```

Listing 2.29: Eine Urlaubsregelung mit elseif

Profitipp

Denken Sie bei Bedingungen immer an alle möglichen Fälle. Wenn Sie in einer *if*-Bedingung auf „kleiner“ prüfen, fehlt neben „größer“ noch der Fall „gleich“. Wenn man keinen Gewinn erwirtschaftet, macht man nicht unbedingt Verlust. Es kann auch sein, dass das Ergebnis 0 lautet. Werden solche Fälle übersehen, erhält man eine Software, die in 99.9 % der Fälle korrekt funktioniert, jedoch „plötzlich“ eine falsche Annahme macht. Sucht man dann die fehlerhafte Verarbeitung, ist das meist sehr zeitaufwendig.

Das Fragezeichen als Verzweigung

Bei vielen Programmiersprachen wie auch in PHP existiert eine weitere *if-then-else*-Konstruktion, die jedoch sehr selten zum Einsatz kommt. Listing 2.30 zeigt das Setzen einer Ausgabe in Abhängigkeit einer Variablen *$alter*, wobei zwischen „jung“ und „alt“ unterschieden werden soll. Das würden Sie bislang mit einer einzigen *if-then-else*-Verzwei-

gung lösen. Der Fragezeichenoperator, der auch als Trinitätsoperator bezeichnet wird, bietet eine Lösung in einer einzigen Zeile bei minimalem zusätzlichen Text:

```
<?php
  $alter=50;
  ($alter<50) ? $ausgabe="jung" : $ausgabe="alt";
?>
<html><body><?php echo $ausgabe?></body></html>
```

Listing 2.30: Eine Fallunterscheidung mit Fragezeichen

Zunächst muss eine Bedingung formuliert werden, die einen Wahrheitswert zurück gibt. Diese Bedingung wird in Klammern gesetzt. Ihr folgt der *?*-Operator. Die Anweisung hinter dem Operator wird genau dann ausgeführt, wenn die Bedingung erfüllt ist, der Wahrheitswert also *TRUE* ist. Ist das Alter also kleiner als 50 Jahre, so wird die Zuweisung *$ausgabe="jung"* ausgeführt. Wenn die Bedingung nicht erfüllt ist, so wird die Anweisung ausgeführt, die hinter dem Doppelpunkt steht, also *$ausgabe="alt"*. Der Doppelpunkt gehört also zu dem Fragezeichenoperator und trennt das *then* von dem *else*.

Meinung

Programmieranfänger halten den Fragezeichenoperator für kryptisch und den erzeugten Quellcode für schlecht lesbar. Profis verwenden gerade diesen Operator jedoch gern, um kompakten Quellcode zu verfassen. Überlegen Sie am besten, wer in Zukunft Einblick in Ihren Quellcode bekommt. Gehen Sie von einem erfahrenen Umfeld von PHP-Entwicklern aus, können Sie den Operator gern verwenden.

Die switch-Verzweigung

Die *switch*-Verzweigung ist nicht so flexibel wie eine *elseif*-Konstruktion, bietet aber eine sehr übersichtliche Struktur für eine Fallunterscheidung einer einzelnen Variable an. Mit dem Namen der Variablen beginnt der *switch*-Block, in dessen Rahmen die verschiedenen Fälle (Cases) abgearbeitet werden. Die Verarbeitung beginnt bei dem ersten zutreffenden *case*-Block und wird bis zum nächsten *break*-Kommando abgearbeitet. Die Verarbeitung kann dadurch über mehrere *case*-Blöcke verlaufen. Abbildung 2.7 zeigt die Arbeitsweise der *switch*-Verzweigung anhand eines Programmablaufplans.

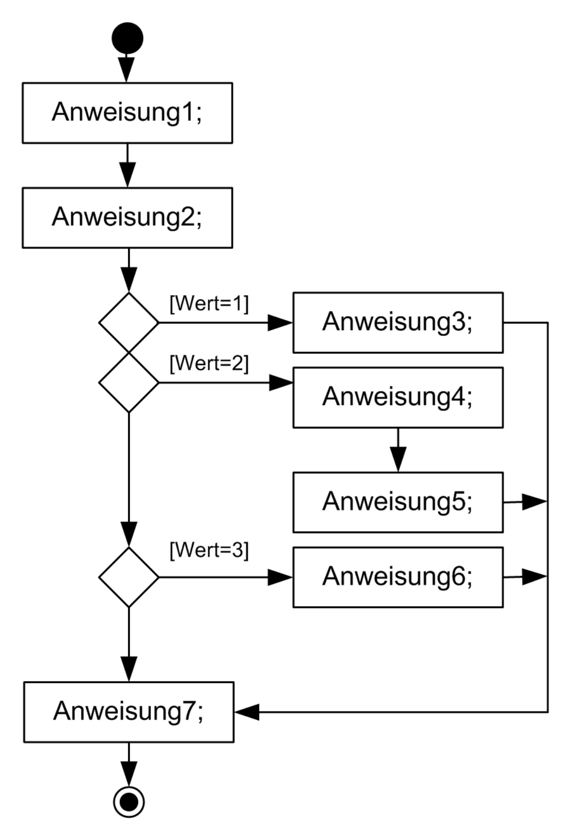


Abbildung 2.7: Die switch-Verzweigung

Im ersten Beispiel erhalten Sie eine Schulnote aus einer Datenbank, die in der Variablen *$note* gespeichert wird. Sie möchten nun für die Ausgabe den Text der Note ermitteln. Die Fälle 4 bis 6 wurden ausgelassen. Wie Sie sehen, erzeugt die *switch*-Anweisung sehr übersichtlichen Quellcode:

```
<?php
  $note=2;
  switch($note){
    case 1:
      $noteText="Sehr gut";
      break;
    case 2:
      $noteText="Gut";
      break;
    case 3:
      $noteText="Befriedigend";
```

Listing 2.31: Eine Note per switch-Anweisung

```
      break;
    // Fälle 4 bis 6
    default:
      $noteText="--ERROR--";
      break;
  }
?>
<html><body>
  Sie haben mit der Note "<?php echo $noteText?>" abgeschlossen!
</body></html>
```

Listing 2.31: Eine Note per switch-Anweisung (Forts.)

Ähnlich wie das letzte *else* in der *elseif*-Struktur existiert bei dem *switch*-Block bei Bedarf ein abschließender *default*-Block. Damit können alle Fälle abgefangen werden, die vorher nicht behandelt wurden. Ob die Verwendung der *switch*-Verzweigung übersichtlicher ist als die *elseif*-Konstruktion, ist Geschmacksache.

Die Urlaubsregelung aus Listing 2.29 kann jedenfalls nicht mit einer *switch*-Verzweigung gelöst werden, da die Urlaubsregelung Wertebereiche des Alters abfragt, beispielsweise „von inklusive 20 Jahren bis 40 Jahren". Das ist mit einem *switch*-Befehl nicht möglich. Die *switch*-Verzweigung ist also in ihrer Mächtigkeit beschränkter als eine *elseif*-Konstruktion.

Sie können jedoch mehrere diskrete Fälle in einem Block abhandeln, wie das Beispiel in Listing 2.32 zeigt. Hier werden die Schulnoten 1 bis 4 sowie 5 und 6 durch Auslassen der *break*-Anweisungen zusammengefasst:

```
<?php
  $note=2;
  switch($note){
    case 1:
    case 2:
    case 3:
    case 4:
      $bestandenText="Sie haben bestanden :-)";
      break;
    case 5:
    case 6:
      $bestandenText="Sie sind durchgefallen :-(";
      break;
    default:
      $bestandenText="--ERROR--";
      break;
```

Listing 2.32: Eine switch-Verzweigung mit zusammengefassten Fällen

```
  }
?>
<html><body><?php echo $bestandenText?></body></html>
```

Listing 2.32: Eine switch-Verzweigung mit zusammengefassten Fällen (Forts.)

2.1.4 Schleifen

Im letzten Kapitel wurden verschiedene PHP-Befehle vorgestellt, um die Ausführung von Quellcode zu verzweigen. Dadurch kann entweder ein Teil A oder ein Teil B eines Programms ausgeführt werden. Mit *elseif* oder *switch* können mehrfache Verzweigungen durchgeführt werden.

In diesem Kapitel wird nun eine weitere, für prozedurale und objektorientierte Sprachen typische Kontrollstruktur vorgestellt. Mit einer Schleife können Sie Quellodeteile so oft wiederholen, wie eine Bedingung erfüllt ist. Generell unterscheidet man drei Arten von Schleifen:

- Kopfgesteuerte Schleifen:
  Dabei wird die Bedingung abgefragt, bevor die Schleife das erste Mal durchlaufen wird. Nach einem einmaligen Durchlauf wird die Bedingung dann nochmals geprüft. Wenn die Bedingung vor dem ersten Durchlauf nicht erfüllt ist, wird die Schleife nicht ausgeführt.
- Fußgesteuerte Schleifen:Zählschleifen:
  Bei den Zählschleifen handelt es sich um eine besondere Form der kopfgesteuerten Schleifen. Eine Zählschleife zählt eine Variable von einem unteren Grenzwert bis zu einem oberen Grenzwert durch. Der Zähler wird dann meist für den Zugriff auf eine Datenstruktur, beispielsweise auf ein Feld, verwendet.

Abbildung 2.8 stellt den Ablauf einer kopf- und fußgesteuerten Schleife gegenüber. Nach Abarbeitung der Anweisung 1 wird die Bedingung der kopfgesteuerten Schleife geprüft. Ist sie erfüllt, werden Anweisung 2 und 3 aus dem Schleifenrumpf einmalig ausgeführt. Danach erfolgt die erneute Prüfung. Bei der fußgesteuerten Schleife werden hingegen die Anweisungen 1 bis 3 auf jeden Fall einmalig ausgeführt. Die Prüfung erfolgt stets nach der Ausführung der Anweisungen im Schleifenrumpf.

Hinweis

Prüfen Sie bei der Programmierung genau, ob die Anweisungen im Schleifenrumpf tatsächlich dafür sorgen, dass die Bedingung irgendwann nicht mehr erfüllt ist. Ist das nicht der Fall, so erzeugen Sie eine Endlosschleife. PHP bricht bei einer Verarbeitungszeit über 30 Sekunden standardmäßig die Verarbeitung ab, um die Last auf dem Server zu verringern.

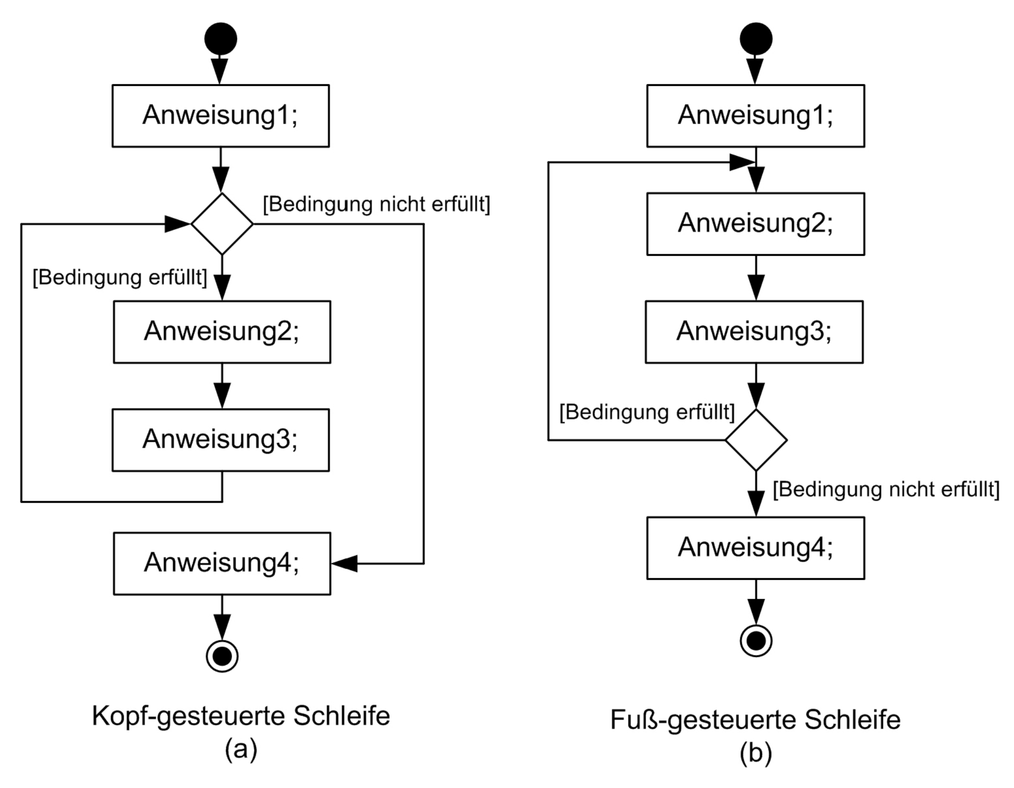


Abbildung 2.8: Ablaufplan einer kopf- (a) und fußgesteuerten (b) Schleife

for-Schleife

Bei der *for*-Schleife handelt es sich um eine kopfgesteuerte Zählschleife, bei der neben der Bedingung noch die Initialisierung der Zählvariablen sowie die Anweisung zum Verändern der Zählvariablen angegeben werden. Das bedeutet, dass die Zählvariable nicht unbedingt um 1 erhöht werden muss, sondern beliebig gemäß der Anweisung verändert werden kann.

Das Beispiel in Listing 2.33 zeigt bereits die Effizienz einer Schleife. Wenn Sie bislang große HTML-Tabellen manuell erstellt haben, ist Ihnen sicherlich der Aufwand der Zeilenbeschreibung aufgefallen. Der resultierende Quellcode kann sehr unübersichtlich werden, insbesondere wenn in einer Tabelle noch zahlreiche CSS-Angaben zur Darstellung enthalten sind.

Listing 2.33 erzeugt eine einfache HTML-Tabelle mit einer Spalte und 10 Zeilen. Die Zählvariable wird typischerweise mit *$i* benannt und durchlüft die Werte 1 bis 10, wobei sie bei jedem Durchlauf mit *$i++* um einen Wert inkrementiert wird. Im Rumpf der Schleife wird der PHP-Code unterbrochen und HTML ausgegeben, nämlich jeweils genau eine Zeile. Als Text innerhalb einer Tabellenzelle wird einfach der aktuelle Wert der Zählvariablen mit *<?php echo $i?>* ausgegeben. Die Definition der Tabelle selbst, also *<table>* und *</table>* liegt außerhalb der Schleife, da diese Definition nur einmalig erfolgt. Der 11-zeilige PHP-Quelltext erzeugt bereits 24 Zeilen HTML-Code in der Ausgabe:

```
<html><body>
  <table border="1">
  <?php
    for($i=1;$i<=10;$i++){
      ?>
        <tr><td align="right"> <?php echo $i?> </td></tr>
      <?php
    }
  ?>
  </table>
</body></html>
```

Listing 2.33: Eine for-Schleife erzeugt eine HTML-Tabelle

Typischerweise besitzt eine Tabelle nicht nur eine einzige Spalte, sondern mehrere. Schleifen können nämlich genauso wie Verzweigungen verschachtelt werden. Dadurch kann ein Zähler *$i* über die Zeilen und ein zweiter Zähler *$j* über die Spalten der Tabelle laufen. Listing 2.34 zeigt eine Tabelle mit 10 Zeilen und 8 Spalten. Die Ausgabe der Tabelle ist in Abbildung 2.9 dargestellt:

```
<html><body>
  <table border="1">
  <?php
    for($i=1;$i<=10;$i++){
      echo('<tr>'); // neue Zeile
      for($j=1;$j<=8;$j++){
        ?><td align="right">Zeile <?php echo $i?>, Spalte <?php echo $j?></td><?php
      }
      echo('</tr>');
    }
  ?>
  </table>
</body></html>
```

Listing 2.34: Eine Tabelle mit 8 Spalten durch verschachtelte for-Schleife

Zeile 1, Spalte 1	Zeile 1, Spalte 2	Zeile 1, Spalte 3	Zeile 1, Spalte 4	Zeile 1, Spalte 5	Zeile 1, Spalte 6	Zeile 1, Spalte 7	Zeile 1, Spalte 8
Zeile 2, Spalte 1	Zeile 2, Spalte 2	Zeile 2, Spalte 3	Zeile 2, Spalte 4	Zeile 2, Spalte 5	Zeile 2, Spalte 6	Zeile 2, Spalte 7	Zeile 2, Spalte 8
Zeile 3, Spalte 1	Zeile 3, Spalte 2	Zeile 3, Spalte 3	Zeile 3, Spalte 4	Zeile 3, Spalte 5	Zeile 3, Spalte 6	Zeile 3, Spalte 7	Zeile 3, Spalte 8
Zeile 4, Spalte 1	Zeile 4, Spalte 2	Zeile 4, Spalte 3	Zeile 4, Spalte 4	Zeile 4, Spalte 5	Zeile 4, Spalte 6	Zeile 4, Spalte 7	Zeile 4, Spalte 8
Zeile 5, Spalte 1	Zeile 5, Spalte 2	Zeile 5, Spalte 3	Zeile 5, Spalte 4	Zeile 5, Spalte 5	Zeile 5, Spalte 6	Zeile 5, Spalte 7	Zeile 5, Spalte 8
Zeile 6, Spalte 1	Zeile 6, Spalte 2	Zeile 6, Spalte 3	Zeile 6, Spalte 4	Zeile 6, Spalte 5	Zeile 6, Spalte 6	Zeile 6, Spalte 7	Zeile 6, Spalte 8
Zeile 7, Spalte 1	Zeile 7, Spalte 2	Zeile 7, Spalte 3	Zeile 7, Spalte 4	Zeile 7, Spalte 5	Zeile 7, Spalte 6	Zeile 7, Spalte 7	Zeile 7, Spalte 8
Zeile 8, Spalte 1	Zeile 8, Spalte 2	Zeile 8, Spalte 3	Zeile 8, Spalte 4	Zeile 8, Spalte 5	Zeile 8, Spalte 6	Zeile 8, Spalte 7	Zeile 8, Spalte 8
Zeile 9, Spalte 1	Zeile 9, Spalte 2	Zeile 9, Spalte 3	Zeile 9, Spalte 4	Zeile 9, Spalte 5	Zeile 9, Spalte 6	Zeile 9, Spalte 7	Zeile 9, Spalte 8
Zeile 10, Spalte 1	Zeile 10, Spalte 2	Zeile 10, Spalte 3	Zeile 10, Spalte 4	Zeile 10, Spalte 5	Zeile 10, Spalte 6	Zeile 10, Spalte 7	Zeile 10, Spalte 8

Abbildung 2.9: Eine dynamisch erzeugte HTML-Tabelle

Im weiteren Verlauf dieses Buchs werden solche Tabellen mit Inhalten aus einer Datenbank, beispielsweise mit einer Auswahl von zu kaufenden Artikeln oder einem Warenkorb-Bestand, gefüllt.

Ein weiterer Anwendungsfall für *for*-Schleifen liegt in dem Zugriff auf Datenfelder. Listing 2.35 gibt die Elemente eines Arrays nacheinander aus und multipliziert deren Inhalt gleichzeitig. Nach der Schleife wird das Produkt der Zahlen dann ausgegeben. Achten Sie darauf, dass die Variable *$produkt* mit 1 initialisiert sein muss:

```
<?php
  $data=Array();
  $data[]=4; $data[]=34; $data[]=546;
?>
<html><body>
  <?php
    $produkt=1;
    for($i=0;$i<count($data);$i++){
      echo(var_dump($data[$i]).'<br>');
      $produkt*=$data[$i];
    }
    echo('Das Produkt ist: '.$produkt);
  ?>
</body></html>
```

Listing 2.35: Zugriff auf ein Datenfeld und Ausführen einer Multiplikation

Der Zugriff auf ein Datenfeld mit einer Zählschleife ist jedoch gerade bei PHP aufgrund der dynamischen Handhabung von Arrays problematisch. Im Beispiel von Listing 2.36 werden die Elemente 1 und 4 im Array *$data* mit Werten gefüllt. Da das Feld nun zwei Elemente besitzt, liefert *count($data)* den Wert 2 zurück. Die *for*-Schleife liest daraufhin *$data[0]=NULL* und *$data[1]=4* aus. Der *NULL*-Wert wird zur Ganzzahl 0 umgewandelt, sodass sich ein Produkt von 0 ergibt. Die Berechnung ist also völlig fehlerhaft. Erfolgt die Zuweisung des Datenfelds an einer anderen Stelle oder sogar in einer anderen Datei, so ist dieser Fehler zunächst sehr schwierig zu finden:

```
<?php
  $data=Array();
  $data[1]=4; $data[4]=546;
?>
<html><body>
  <?php
    echo('Anzahl Elemente: '.count($data).'<br>';
    $produkt=1;
    for($i=0;$i<count($data);$i++){
```

Listing 2.36: Falsche Berechnung des Produkts

```
      echo(var_dump($data[$i]).'<br>');
      $produkt*=$data[$i];
    }
    echo('Das Produkt ist: '.$produkt);
  ?>
</body></html>
```

Listing 2.36: Falsche Berechnung des Produkts (Forts.)

Profitipp

Obwohl die *for*-Zählschleife in anderen Sprachen oft zu Operationen auf Datenfelder angewendet wird, ist bei PHP davon abzuraten. Der Grund liegt in der dynamischen Zuweisung von Feldern, bei denen Elemente innerhalb des Felds unter Umständen nicht mit Werten belegt wurden. Das ist eine ernstzunehmende Fehlerquelle in PHP-Skripten. Ebenso eignet sich die Zählschleife nicht für assoziative Arrays. In Verbindung mit Datenfeldern sollte daher ausschließlich die *foreach*-Schleife aus dem nächsten Kapitel verwendet werden.

foreach-Schleife

Die kopfgesteuerte *foreach*-Schleife löst auf einfachem Wege den Zugriff auf teil-initialisierte oder assoziative Datenfelder, indem nur die befüllten Elemente eines Arrays nacheinander zugegriffen werden. Die *foreach*-Schleife speichert dabei den Index und den aktuellen Wert des Elements, das gerade untersucht wird. Listing 2.37 zeigt die verbesserte Berechnung des Produkts im Vergleich zu Listing 2.36:

```
<?php
  $data=Array();
  $data[1]=4; $data[4]=546;
?>
<html><body>
  <?php
    $produkt=1;
    foreach ($data as $index => $wert){
      echo('Index: '.$index.', Inhalt: '.$wert.'<br>');
      $produkt*=$wert;
    }
    echo('Das Produkt ist: '.$produkt);
  ?>
</body></html>
```

Listing 2.37: Korrigierte Berechnung des Produkts mit foreach-Schleife

Auf eine Zählvariable wird bei der *foreach*-Schleife verzichtet. Dadurch werden Endlosschleifen durch falsche Inkrementierung und Verletzungen der Datenfeldgrenzen ver-

mieden. Stattdessen wird als erster Parameter das zu untersuchende Datenfeld, hier: *$data*, angegeben. Hinter dem Schlüsselwort *as* wird das aktuelle Element des Felds in dessen Index und dessen Wert aufgespaltet. Im Rumpf der Schleife kann man nun auf diese beiden Variablen zugreifen, sodass die folgende korrekte Ausgabe entsteht:

Index: 1, Inhalt: 4

Index: 4, Inhalt: 546

Das Produkt ist: 2184

Auf die gleiche Art und Weise kann ein assoziatives Datenfeld ausgegeben werden. Außerdem können auch mehrere *foreach*-Schleifen geschachtelt werden, wie Listing 2.38 zeigt. Die Datenstruktur der Kundenliste wurde aus Listing 2.18 entnommen:

```
<html><body>
  <?php
    $kunden=array(
      'K1'=>array('VN'=>'Frank','N'=>'Dopatka'),
      'K2'=>array('VN'=>'Uli','N'=>'Müller'),
      'K3'=>array('VN'=>'Max','N'=>'Mustermann')
    );
    foreach ($kunden as $kundennr => $kunde){
      echo('Kunden-Nr.: '.$kundennr.'<br>');
      foreach ($kunde as $index => $wert){
        echo($index.': '.$wert.'<br>');
      }
    }
  ?>
</body></html>
```

Listing 2.38: Ausgabe eines zweidimensionalen Feldes mit geschachtelter foreach-Schleife

In der äußeren *foreach*-Schleife wird die Kundennummer und der Kunde voneinander getrennt in den Variablen *$kunndennr* und *$kunde*. Bei *$kunde* handelt es sich wiederum um ein Datenfeld, das in der inneren *foreach*-Schleife abgearbeitet wird. Dort werden der Vorname und der Name getrennt und ausgegeben, sodass folgende Gesamtausgabe im Internetbrowser des Clients entsteht:

Kunden-Nr.: K1

VN: Frank

N: Dopatka

Kunden-Nr.: K2

VN: Uli

N: Müller

Kunden-Nr.: K3

VN: Max

N: Mustermann

Als Übung können Sie die das Datenfeld *$kunden* um einige Elemente erweitern und die Ausgabe in eine HTML-Tabelle wie in Listing 2.34 umstrukturieren.

while-Schleife

Als nächsten Schleifentyp wird die kopfgesteuerte *while*-Schleife vorgestellt. Hinter dem Schlüsselwort *while* wird lediglich eine Bedingung in runden Klammern angegeben. Ist die Bedingung erfüllt, wird der Rumpf der Schleife genau einmal durchlaufen. Danach wird die Bedingung nochmals geprüft. Innerhalb der Schleife muss nach einem x-ten Durchlauf die Bedingung so verändert worden sein, dass sie *FALSE* zurück gibt. Dann wird die Schleife verlassen.

Listing 2.39 zeigt eine *while*-Schleife, die eine Variable *$a* bis zu einer Obergrenze, die in einer zweiten Variablen *$b* gespeichert ist, hochzählt. Das lässt sich natürlich auch mit einer *for*-Schleife realisieren. Generell lässt sich jede *while*-Schleife in eine *for*-Schleife umwandeln und umgekehrt. Es handelt sich lediglich um eine andere Syntax.

Eine *while*-Schleife wird jedoch erfahrungsgemäß eher eingesetzt, wenn die Grenzen nicht im Vorfeld bekannt sind oder wenn die Bedingung der Schleife komplex ist:

```
<html><body>
  <?php
    $a=12;
    $b=23;
    while ($a<$b){
      echo('Der Wert '.$a.' ist kleiner als '.$b.'<br>'); $a++;
    }
  ?>
</body></html>
```

Listing 2.39: Beispiel einer while-Schleife

Im zweiten Beispiel wird ein Datenfeld mit einem Index von 0 bis 99 über eine *while*-Schleife mit einer mathematischen Formel befüllt. Mithilfe einer zweiten *while*-Schleife wird das Feld dann solange ausgelesen, solange der Feldinhalt kleiner als 200 ist. Die ausgegebenen Werte lauten 0, 6, 26, 60, 108 und 170. Wenn Sie in einem Fall die Inkrementoperation *$i++* vergessen, erhalten Sie eine Endlosschleife:

```
<html><body>
  <?php
    $data=Array();
    $i=0;
```

Listing 2.40: Füllen eines Datenfelds und dessen Auslesen mit zwei while-Schleifen

```
    while($i<100){
      $data[$i]=7*$i*$i-$i; $i++;
    }
    $i=0;
    while($data[$i]<200){
      echo('Wert: '.$data[$i].'<br>'); $i++;
    }
  ?>
</body></html>
```

Listing 2.40: Füllen eines Datenfelds und dessen Auslesen mit zwei while-Schleifen (Forts.)

Im weiteren Verlauf dieses Buchs werden *while*-Schleifen insbesondere beim Einlesen von Datenquellen wie Dateien oder Datenbankeinträgen verwendet. Die Dateien werden zeilenweise eingelesen und zwar solange, bis das Dateiende erreicht ist. Das bildet die Bedingung für die *while*-Schleife.

Bei einer Datenbank wird eine SQL-Abfrage abgesetzt, die eine beliebig große Ergebnismenge in einer zweidimensionalen Tabelle zurück gibt. Diese Ergebnismenge wird dann zeilenweise bis zum Ende durchlaufen und die Daten interpretiert. Auch dafür eignet sich eine *while*-Schleife besonders gut.

do-while

Die *do-while*-Schleife ist die einzige fußgesteuerte Schleife in PHP, die mindestens einmal durchlaufen wird. Am Ende des Schleifenrumpfes wird dann die Bedingung geprüft. Liefert die Bedingung *TRUE* zurück, so wird die Schleife nochmals durchlaufen.

Das Beispiel in Listing 2.41 zeigt den Aufbau der Fibonacci-Folge in einem Datenfeld und dessen Ausgabe mit einer zweiten *do-while*-Schleife:

```
<html><body>
  <?php
    $fib=Array();
    $obergrenze=10;
    $fib[0]=0; $fib[1]=1;
    $i=2;
    do{
      $fib[$i]=$fib[$i-1]+$fib[$i-2]; $i++;
    }while($i<=$obergrenze);
    $i=0;
    do{
      echo('Fib['.$i.'] ergibt '.$fib[$i].'<br>'); $i++;
    }while($i<=$obergrenze);
```

Listing 2.41: Aufbau und Ausgabe der Fibonacci-Folge mit zwei do-while-Schleifen

```
  ?>
</body></html>
```

Listing 2.41: Aufbau und Ausgabe der Fibonacci-Folge mit zwei do-while-Schleifen (Forts.)

Diese Schleife wird unter anderem für Benutzereingaben verwendet, die mindestens einmal getätigt werden müssen, bis eine besondere Eingabe als „Fertig"-Kennzeichnung erfolgt.

Wie auch die *while*-Schleife kann die *do-while*-Schleife geschachtelt werden, um komplexere Abläufe durchzuführen. Auch hier ist wieder eine Umwandlung in einen anderen Schleifentyp möglich. Welche Schleife letztlich verwendet wird, liegt an Ihrem eigenen Ermessen und Programmierstil. Wenn Sie sich gerade in Ihre erste Programmiersprache einarbeiten, sollten Sie versuchen, die Listings 2.33 bis 2.41 in diesem Kapitel mit anderen Schleifen zu realisieren. Den Erfolg können Sie anhand der Ausgaben prüfen, die bei einer korrekten Umwandlung identisch sein müssen.

Ebenso können Schleifen verschiedenen Typs in Kombination mit Verzweigungen eingesetzt werden. Das Ergebnis ist dann Ihr prozedurales Programm.

Break und continue

Zum Anschluss dieses Kapitels müssen noch zwei besondere PHP-Befehle in Verbindung mit Schleifen erwähnt werden. Bereits in Verbindung mit der *switch*-Verzweigung wurde das *break*-Kommando verwendet, um einen behandelten Fall abzuschließen. Diesen Befehl können Sie auch innerhalb eines Schleifenrumpfes verwenden. Wird er ausgeführt, so wird das Durchlaufen der Schleife sofort beendet und die erste Anweisung hinter dem Schleifenrumpf ausgeführt.

Auf den ersten Blick stellt sich die Frage, welchen Sinn eine Schleife noch macht, wenn man sie mit einem anderen Befehl abbricht. Die Antwort liegt in der Verbindung des *break*-Kommandos mit einer Verzweigung. Ein sinnvolles Beispiel dafür ist die Suche in einem Datenfeld. Die Schleife durchsucht ein Datenfeld nach einem bestimmten Element. Wurde das Element gefunden, muss das Feld nicht weiter durchsucht werden, da es unter Umständen sehr viele Elemente enthalten kann:

```
<html><body>
  <?php
    $data=Array();
    $data[]="Hugo"; $data[]="Uli"; $data[]="Frank"; $data[]="Olga";
    $suche="Frank";
    $gefunden=FALSE;
    foreach($data as $index => $wert){
      if($wert==$suche){
        echo($wert.' gefunden!<br>');
        $gefunden=TRUE; break;
      }
```

Listing 2.42: Eine Suchfunktion in einem Datenfeld und das break-Kommando

```
      else{
        echo('Der Name '.$wert.' ist NICHT der Suchbegriff...<br>');
      }
    }
    if ($gefunden){
      echo('Suche war erfolgreich!<br>');
    }
  ?>
</body></html>
```

Listing 2.42: Eine Suchfunktion in einem Datenfeld und das break-Kommando (Forts.)

Die Ausgabe des Listings lautet:

Der Name Hugo ist NICHT der Suchbegriff...

Der Name Uli ist NICHT der Suchbegriff...

Frank gefunden!

Suche war erfolgreich!

Diese Funktion muss in PHP jedoch nicht über eine zeitaufwendige Schleife realisiert werden, obwohl das im Alltag oft geschieht. Tabelle 2.10 beschreibt bereits die Funktion *in_array*, die in einem einzigen Aufruf das Datenfeld prüft. Listing 2.43 zeigt, wie die Funktionalität wesentlich kompakter und performanter realisiert werden kann:

```
<html><body>
  <?php
    $data=Array();
    $data[]="Hugo"; $data[]="Uli"; $data[]="Frank"; $data[]="Olga";
    $suche="Frank";
    $gefunden=in_array($suche, $data);
    if ($gefunden){
      echo($wert.' gefunden!<br>');
      echo('Suche war erfolgreich!<br>');
    }
  ?>
</body></html>
```

Listing 2.43: Eine verbesserte Suchfunktion

Der PHP-Befehl *continue* bricht im Gegensatz zu *break* nicht die ganze Schleife ab, sondern beendet nur den aktuellen Schleifendurchlauf. Das hat zur Folge, dass unmittelbar nach einem *continue*-Kommando die Bedingung der Schleife wieder geprüft wird. Dann wird entschieden, ob die Schleife ein weiteres Mal durchlaufen wird oder nicht. Auch hier ist der Befehl nur in Verbindung mit einer Verzweigung sinnvoll, mit deren Bedingung das Überspringen des jetzigen Schleifendurchlaufs gesteuert wird.

Im Beispiel des Listings 2.44 wird eine Menge von Ganzzahlen in einer *for*-Schleife durchlaufen. Die Bedingung der Verzweigung innerhalb der *for*-Schleife entscheidet dann, ob eine Ausgabe stattfindet oder nicht. In diesem Beispiel werden nur ungerade Zahlen ausgegeben:

```
<html><body>
  <?php
    for($i=0;$i<50;$i++){
      if ($i%2==0) continue;
      echo($i.'<br>');
    }
  ?>
</body></html>
```

Listing 2.44: Ausgabe der Elemente eines Felds beschränken mit continue

Auch in diesem Fall lässt sich die Lösung eleganter ohne den *continue*-Befehl formulieren, indem der Schleifenkopf zu *for($i=0;$i<50;$i+=2)* geändert wird. Auf diese Weise verhindern Sie unnötige Schleifenprüfungen von geraden Zahlen.

2.1.5 Funktionen

In den bisherigen Beispielen haben Sie bereits mehrfach Funktionen verwendet, ohne dass bislang die Verwendung von Funktionen in PHP genau beschrieben wurde. Ein Beispiel ist der Aufruf *$gefunden=in_array($suche, $data)* in Listing 2.43. Wie eine mathematische Funktion *a=f(x,y)* bekommt die oben genannte PHP-Funktion zwei Parameter als Eingabe und liefert einen Parameter als Ausgabe, der in der Variablen *$gefunden* gespeichert wird. In der Objektorientierung werden Funktionen auch als Methoden oder Operationen bezeichnet.

PHP liefert bereits eine Vielzahl an Funktionen, die Sie verwenden können. Dazu gehören Funktionen zum Zugriff auf Datenfelder, für mathematische Berechnungen, für den Zugriff auf Zeichenketten, zum Rechnen mit Datums- und Zeitwerten und vieles mehr.

Zusätzlich dazu können Sie eigene Funktionen definieren, die Sie mehrfach weiterverwenden können. Das dient dazu, gleichbleibende Teile des Quellcodes nicht mehrfach zu schreiben und verbessert die Wartbarkeit Ihrer Anwendung. Gleichzeitig können Sie die korrekte Funktionsweise einmalig ordentlich testen und danach (relativ) sicher sein, dass Ihre Funktion auch in einem anderen Kontext korrekt funktioniert.

Listing 2.45 zeigt den ersten Versuch, eine Funktion selbst zu schreiben. Ihr Aufruf soll den Wert der Variablen *$a* mit einer zusätzlichen Zeichenkette verknüpft ausgeben. Die Rückgabe einer Funktion definieren Sie durch den *return*-Befehl. Beinhaltet die Funktion weiteren Quellcode hinter einem *return*-Befehl, wird dieser nicht mehr ausgeführt. Der Anwender wird also nicht „zu spät" in seinem Internetbrowser lesen können. Dennoch entspricht die Ausgabe nicht dem erhofften Ergebnis. Lediglich „Welt!" erscheint auf dem Bildschirm:

```
<?php
  $a='Hallo';
  function Ausgabe(){
    return $a.' Welt!';
    echo('zu spät');
  }
?>
<html><body>
  <?php echo Ausgabe()?>
</body></html>
```

Listing 2.45: Keine erfolgreiche Ausgabe durch die Funktion

Die Ursache dafür liegt darin, dass die Variable *$a* nicht innerhalb der Funktion *Ausgabe* definiert ist, sondern lediglich außerhalb. Wenn Sie dennoch auf die externe Variable zugreifen wollen, müssen Sie diese zunächst innerhalb der Funktion bekannt machen. Das geschieht über den *global*-Befehl. Danach erfolgt die Ausgabe „Hallo Welt!“ wie erwartet:

```
<?php
  $a="Hallo";
  function Ausgabe(){
    global $a;
    return $a.' Welt!';
  }
```

Listing 2.46: Korrekte Ausgabefunktion

Wert- und Referenzübergabe von Parametern

In einer strukturierten Anwendung sollte nach Möglichkeit auf die Verwendung von globalen Variablen verzichtet werden. Ein besserer Programmierstil liegt darin, die innerhalb der Funktion benötigten Parameter von außen zu übergeben und das Ergebnis zurück zu geben. Ein Beispiel ist die Funktion *Swap* in Listing 2.47, die zwei Werte vertauschen soll. Diese Werte heißen innerhalb der Funktion *$x* und *$y*.

Leider funktioniert diese Funktion auch nicht auf Anhieb. Die Ausgabe der Werte im HTML-Teil der Datei ist unverändert *$a=33* und *$b=99*. Die Funktion selbst ist jedoch korrekt: Der Wert der x-Variable wird in eine temporäre Variable übergeben. Dann wird der x-Wert vom y-Wert überschrieben und abschließend wird der temporäre Wert in die y-Variable geschrieben:

```
<?php
  $a=33; $b=99;
  function Swap($x,$y){
    $temp=$x; $x=$y; $y=$temp;
```

Listing 2.47: Kein Tausch durch den Swap-Befehl

```
  }
?>
<html><body>
  <?php
    Swap($a,$b);
    echo($a.'<br>'); echo($b.'<br>');
  ?>
</body></html>
```

Listing 2.47: Kein Tausch durch den Swap-Befehl (Forts.)

Die Ursache für den Fehler liegt in der Wertübergabe beim Aufruf der Swap-Funktion. Im Speicher werden nämlich die Werte der Variablen *$a* und *$b* kopiert und ab diesem Zeitpunkt innerhalb der Funktion unter dem Namen *$x* und *$y* weiter verarbeitet.

Stattdessen müssten die Parameter jedoch als Referenz übergeben werden. Dadurch würde *$x* auf dieselbe Speicherstelle zeigen wie *$a* und *$y* auf dieselbe Speicherzelle wie *$b*. Die Unterschiede zwischen Werten und Referenzen wurden bereits in Kapitel 2.1.1 „Verwaltung von Variablen und Referenzen/Zeigern“ vorgestellt.

Um statt einer Kopie der Werte die Referenzen zu übergeben, müssen Sie in der Deklaration der Funktion lediglich den Referenzoperator „&“ vor jeden Parameter setzen, bei dem lediglich eine Referenz übergeben werden soll. Die Funktionsdeklaration lautet dann *function Swap(&$x,&$y)*. Danach funktioniert die Funktion wie erwartet.

Funktionen als Übergabeparameter

Bereits in Kapitel 2.1.2 „PHP-Funktionen zur Bearbeitung von Datenfeldern“ wurden PHP-eigene Befehle vorgestellt, mit denen man eine eigene Funktion auf ein Array anwenden kann. Diese Befehle konnten jedoch noch nicht getestet werden, da Sie noch keine eigene Funktion definieren konnten. Das ist jetzt anders.

Listing 2.48 zeigt am Beispiel des Befehls *array_walk*, wie Sie eine eigene Funktion auf jedes Element eines Datenfeldes anwenden. Die eigene Funktion *addiere* erhält einen Parameter *$x* per Referenz übergeben und addiert einen festen Wert auf diesen Parameter.

Im Gegensatz zu anderen Sprachen müssen Sie bei PHP nicht mit einer Schleife über das Datenfeld laufen, um den Wert zu addieren. Stattdessen können Sie die PHP-Funktion *array_walk* verwenden, die ihrerseits wiederum zwei Parameter benötigt. Der erste Parameter ist das Array und der zweite der Name der Funktion als Zeichenkette:

```
<?php
  $data=Array();
  $data[]=6; $data[]=675; $data[]=46; $data[]=235;
  function addiere(&$x){
    $x=$x+11;
  }
```

Listing 2.48: Eine Funktion wird auf ein Array angewendet

```
  array_walk($data, 'addiere');
?>
<html><body>
  <?php echo var_dump($data);?>
</body></html>
```

Listing 2.48: Eine Funktion wird auf ein Array angewendet (Forts.)

Das Skript arbeitet wie erwartet und addiert die Zahl 11 auf jedes Element des Datenfeldes, das abschließend ausgegeben wird:

array(4) { [0]=> int(17) [1]=> int(686) [2]=> int(57) [3]=> int(246) }

Der Tod des Programms: die

Mit dem Befehl *die* können Sie die Bearbeitung des aktuellen PHP-Skripts unverzüglich beenden. Dem Befehl können Sie als Parameter eine Zeichenkette übergeben, die zum Abschluss noch gesendet wird. Der *die*-Befehl wird meist bei schweren Fehlern verwendet, bei denen eine weitere Abarbeitung des Skripts keinen Sinn mehr machen würde, beispielsweise bei einem fehlgeschlagenen Versuch, eine Datenbankverbindung herzustellen oder eine Datei zu öffnen:

```
<?php
  $db=NULL; // Datenbankverbindung existiert nicht
  if ($db===NULL){
    die('<html><body><h2>Keine Verbindung!<h2></body></html>');
  }
  echo('Hallo?');
?>
<html><body><h2>Datenbank geöffnet.</h2></body></html>
```

Listing 2.49: Das Ende eines Skripts

In Listing 2.49 wird von einer Datenbankverbindung ausgegangen, die in der Variablen *$db* gespeichert ist. Diese Variable ist zurzeit *NULL*, da wir noch keine Verbindung zu einer Datenbank aufbauen können. In diesem Fall führt die Verzweigung zum *die*-Befehl.

Sobald der PHP-Interpreter den *die*-Befehl interpretiert, wird der Inhalt des Parameters in der *die*-Funktion an den Internetbrowser des Clients gesendet. Die Verarbeitung des Skripts endet dann, sodass die Meldung „Hallo?“ nicht mehr ausgegeben wird.

Mit objektorientierten Methoden steht Ihnen über die *try-catch*-Konstuktion eine wesentlich komplexere Fehlerbehandlung zur Verfügung, die im vierten Kapitel dieses Buchs erläutert wird.

Dateien einbinden mit require und include

Statische HTML-Seiten haben den Nachteil, dass sie nicht aus anderen HTML-Seiten zusammengesetzt werden können. Es gibt also keinen HTML-Befehl, der eine andere

HTML-Datei in diese Datei vollständig integriert und das Ergebnis dann zum Internetbrowser zurück sendet. Eine solche Funktionalität ist jedoch sehr nützlich, da eine Homepage ein einheitliches Layout und eine einheitliche Menüführung besitzen sollte. Andererseits möchte man bei einer Änderung im Menü der Homepage nicht jede HTML-Seite ändern. Außerdem ist es sinnvoll, alle notwendigen JavaScript-Funktionen und Meta-Tags einmalig zentral zu verwalten, da bei einer Codeänderung im Alltag oft einige Stellen übersehen werden.

In der Vergangenheit wurden einmalige Titel und Menüleisten meist über Frames angeordnet, um das Problem zu lösen. Der Einsatz von Frames in Homepages gilt jedoch als veraltet, da er andere Probleme mit sich bringt. So können Unterseiten schlecht als Bookmarks im Browser festgehalten werden, Suchmaschinenroboter haben Probleme, den Inhalt einer Seite zu ermitteln und die Barrierefreiheit wird erschwert. So lesen Screenreader Frames stets nacheinander vor und Textbrowser von Personen mit Sehstörungen können Frames meist gar nicht darstellen.

Wenn Sie also mehrere Seiten einheitlich verwalten möchten, benötigen Sie aus heutiger Sicht eine andere Lösung wie ein Content-Management-System (CMS), mit dem Sie den Inhalt und die Menüführung verwalten können. Bekannte Content-Management-Systeme wie Typo3 oder Drupal lösen diese Probleme, sind jedoch selbst in PHP geschrieben. PHP besitzt also eine Möglichkeit, wie gewünschte Modularität zu realisieren. Das geschieht insbesondere durch die Verwendung folgender Befehle:

- *require('datei.php')*
- *require_once('datei.php')*
- *include('datei.php')*
- *include_once('datei.php')*

Alle Befehle binden Quelltext aus einer anderen Datei in das aktuelle Skript ein. Der Unterschied zwischen den *include*- und *require*-Befehlen liegt in dem Umgang mit Fehlern, insbesondere wenn die einzubindende Datei nicht existiert. Während *include* eine PHP-Warnmeldung zurückgibt, erzeugt *require* einen PHP-Fehler, wonach die Verarbeitung des Skripts abgebrochen wird. Verwenden Sie also *require*, wenn Sie möchten, dass eine fehlende Datei die Ausführung ihres Skripts beendet.

Ein Problem kann sich ergeben, wenn PHP-Dateien sich gegenseitig mehrfach includieren. Wird beispielsweise eine PHP-Datei mehrfach eingebunden, die eine Funktionsdefinition enthält, dann führt das zu einem PHP-Fehler. Das können Sie mit *require_once* bzw. mit *include_once* verhindern. In diesem Fall prüft PHP vor der Einbindung der Fremddatei, ob diese bereits eingebunden ist. Wenn das der Fall ist, wird sie nicht nochmals eingebunden.

Abschließend wird der *include*-Befehl getestet. Eine Homepage soll dabei in einen einheitlichen Header und Footer strukturiert werden. Der Header enthält alle PHP-Initialisierungen, Skripte, Funktionsdefinitionen und Meta-Tags. Der Footer soll aus einer Copyrightmeldung bestehen. Listing 2.50 zeigt den Quelltext, der dann noch übrig bleibt und der sich auf den eigentlichen Content der Webseite beschränkt:

```
<?php include_once('header.inc.php'); ?>
<h2>Herzlich Willkommen!</h2>
<p>Dies ist der Content dieser Web-Seite. Ist der HTML-Code nicht sehr
   übersichtlich gehalten? Das kann doch jeder editieren, oder?</p>
<?php include_once('footer.inc.php'); ?>
```

Listing 2.50: Content-Datei mit Einbindung eines Headers und eines Footers

Um eine vollständige HTML-Seite zu erhalten, müssen Sie jetzt noch die Datei *header.inc.php* in dasselbe Verzeichnis schreiben:

```
<?php
  $wert=100; // auch alle Initialisierungen
?>
<html><head>
  <meta name="author" content="Frank Dopatka">
  <meta name="copyright" content="Frank Dopatka">
  <meta name="language" content="de">
  <title>Dr. Ds Homepage</title>
</head>
<body>
```

Listing 2.51: Die header.inc.php

Alle Tags, die im Header geöffnet werden und noch nicht geschlossen sind, sollten im Footer geschlossen werden. Dadurch wird Konsistenz für den Entwickler des Content-Teils erreicht, der dann nur die Tags schließen muss, die er auch selbst öffnet. In diesem Header bleiben die Tags *<html>* und *<body>* geöffnet. Diese werden vom Footer in Listing 2.52 geschlossen. Zusätzlich wird der Copyrighthinweis hinzugefügt:

```
  <center>
    Copyright by Dr. Frank Dopatka, 2009
  </center>
</body>
</html>
```

Listing 2.52: Die footer.inc.php

Die drei Dateien werden serverseitig zusammengesetzt und zum Client übertragen. Dieser erhält als Ergebnis eine korrekte HTML-Datei zur Ausgabe.

Profitipp

Es macht keinen Sinn, die PHP-Dateien des Headers und Footers direkt im Internetbrowser aufzurufen. Diese Dateien sind ausschließlich dazu bestimmt, von anderen PHP-Skripten eingebunden zu werden. Es hat sich eingebürgert, solche Dateien mit der Endung *.inc.php* zu versehen, damit sie leichter von anderen Entwicklern erkannt werden können.

Funktionen zur Bearbeitung von Zeichenketten

PHP hat den Ruf, umfangreiche und leicht anwendbare Funktionen zur Bearbeitung von Zeichenketten zu besitzen, insbesondere in Bezug auf Internetanwendungen. In diesem Kapitel werden nur die wichtigsten Funktionen vorgestellt, die im Alltag Anwendung finden. Weitere Funktionen können Sie mithilfe von Internetsuchmaschinen und PHP-Portalen wie *http://www.phpbox.de/php_befehle/zeichenketten.php* ermitteln. Für die selbst erstellten Anwendungen haben sich die im Folgenden aufgeführten Befehle bereits als absolut ausreichend erwiesen.

Tabelle 2.12 zeigt die wichtigsten Funktionen zur Ermittlung der Anzahl von Zeichen in einer Zeichenkette sowie zum Suchen und Vergleichen von Zeichenketten.

Funktion	Bedeutung
$wert=strlen($str)	gibt die Anzahl der Zeichen in *$str* zurück
$wert=strpos($str,$such,$offset)	gibt die erste Position von $such in der Zeichenkette *$str* ab dem Wert von *$offset* zurück
$wert=strrpos($str,$such)	gibt die letzte Position von *$such* in der Zeichenkette *$str* zurück
$erg=strstr($str,$such)	sucht *$such* in der Zeichenkette *$str* und gibt die Teilzeichenkette von *$str* ab der gefundenen Position bis zum Ende zurück
$erg=substr($str,$start,$len)	gibt die Teilzeichenkette ab der Position *$start* von *$str* mit der Länge *$len* zurück
$erg=strcmp($str1,$str2)	vergleicht *$str1* und *$str2* und gibt -1 zurück, wenn *$str1* < *$str2*, 0 wenn beide Strings gleich sind und +1, wenn *$str1* > *$str2*
$erg=strcasecmp($str1,$str2)	wie *strcmp*, berücksichtigt jedoch keine Groß- und Kleinschreibung

Tabelle 2.12: Funktionen zum Suchen und Vergleichen von Zeichenketten

Listing 2.53 zeigt Tests der Zeichenkettenfunktionen. Die Ausgaben lauten „6“, „2“, „llo liebe Leute.“, „liebe“ und „-1“:

```
<html><body>
<?php
  echo(strlen("Hallo?").'<br>');
  $wert=strpos("Hallo?","l",1);
  echo($wert.'<br>');
```

Listing 2.53: Test der Funktionen zum Suchen und Vergleichen von Zeichenketten

```
  echo(strstr("Hallo liebe Leute.","l").'<br>');
  echo(substr("Hallo liebe Leute.",6,5).'<br>');
  echo(strcmp("Frank","Hans").'<br>');
?>
</body></html>
```

Listing 2.53: Test der Funktionen zum Suchen und Vergleichen von Zeichenketten (Forts.)

Die zweite Sammlung von Funktionen manipuliert und ersetzt Zeichenketten. Die *trim*-Funktionen werden besonders bei der Auswertung von Benutzereingaben oder beim Auslesen aus Werten einer Datenbank verwendet, um überflüssige Leerzeichen zu eliminieren, die ansonsten auch eine Prüfung auf Gleichheit zweier Zeichenketten erschweren.

Funktion	Bedeutung
$erg=addcslashes($str,$charlist)	setzt C-typische Escape-Zeichen vor jedem Sonderzeichen, dass in *$charlist* angegeben ist und gibt den
$erg=stripcslashes($str,$charlist)	entfernt C-typische Escape-Zeichen vor jedem Sonderzeichen, dass in *$charlist* angegeben ist
$erg=addslashes($str)	setzt einen Backslash vor speziellen Sonderzeichen
$erg=stripslashes($str)	entfernt den gesetzten Backslash vor speziellen Sonderzeichen
$erg=ltrim($str)	entfernt führende Leerzeichen
$erg=rtrim($str)	entfernt nachfolgende Leerzeichen
$erg=trim($str)	entfernt alle Leerzeichen am Anfang und Ende von *$str*
$erg=str_replace($such,$ers,$str)	ersetzt in *$str* jedes Vorkommen von $such durch *$ers*

Tabelle 2.13: Funktionen zum Ersetzen von Zeichen in Zeichenketten

Listing 2.54 testet einige dieser Funktionen und führt zu folgender Ausgabe:

Hall\366chen an '\326si' \374ber den Bergpa\337!

Hallöchen an \'Ösi\' über den Bergpaß!

Hallöchen an 'Frank' über den Bergpaß!

```
<html><body>
<?php
  $str="Hallöchen an 'Ösi' über den Bergpaß!";
  echo(addcslashes($str,"Ööüß").'<br>');
  echo(addslashes($str).'<br>');
  echo(str_replace("Ösi","Frank",$str));
?>
</body></html>
```

Listing 2.54: Test der Funktionen zum Ersetzen von Zeichen

Tabelle 2.14 zeigt Funktionen, die Zeichen oder Zeichenketten umwandeln. Bei dem Vergleich eines eingegebenen Benutzernamens mit einem Benutzer-Eintrag aus einer Datenbank wird beispielsweise gern die Funktion *strtolower* auf beiden Seiten des Gleich-Operators verwendet. Dadurch ist die Eingabe des Benutzernamens nicht case-sensitiv.

Die Funktionen *implode* und *explode* zur Umwandlung von Zeichenketten in Datenfelder und umgekehrt werden oft beim Laden und Speichern von Daten in Dateien verwendet.

Funktion	Bedeutung
$erg=strrev($str)	invertiert die Zeichenkette *$str*
$erg=strtolower($str)	wandelt *$str* in Kleinbuchstaben um
$erg=strtoupper($str)	wandelt *$str* in Großbuchstaben um
$erg=ord($char)	gibt den ASCII-Wert des Zeichens zurück
$char=chr($byte)	gibt das Zeichen des eingegebenen ASCII-Werts zurück
$arr=explode($sep, $str)	trennt *$str* anhand von *$sep* aus und liefert ein Datenfeld zurück
$str=implode($sep, $arr)	wandelt ein Datenfeld in eine Zeichenkette um und fügt zwischen den Elementen den Separator *$sep* ein

Tabelle 2.14: Funktionen zur Umwandlung von Zeichenketten

Listing 2.55 testet die Funktionen und führt zu folgender Ausgabe:

!etueL ella na ollaH

hallo an alle leute!

HALLO AN ALLE LEUTE!

70

W

Hallo - an - alle - Leute! -

```
<html><body>
<?php
  $str="Hallo an alle Leute!";
  echo(strrev($str).'<br>');
  echo(strtolower($str).'<br>');
  echo(strtoupper($str).'<br>');
  echo(ord("F").'<br>');
  echo(chr(87).'<br>');
  $arr=explode(" ", $str);
  foreach ($arr as $index => $wert){
    echo($wert.' - ');
  }
```

Listing 2.55: Test der Umwandlung von Zeichenketten

```
?>
</body></html>
```

Listing 2.55: Test der Umwandlung von Zeichenketten (Forts.)

Abschließend sind noch einige besondere Funktionen zur Verwendung von Zeichenketten in Internetanwendungen zu nennen. So wandelt z. B. die Funktion *nl2br* die von einer Datei eingelesenen Zeilenumbrüche in <*br*>-Tags um, sodass die Zeilenumbrüche auch bei einer Interpretation des entstehenden HTML-Codes erhalten bleiben.

Eine ähnliche Transformation führt der Befehl *htmlentities* durch. Hier werden Sonderzeichen so umgeformt, dass sie bei der HTML-Interpretation identisch auf dem Browser dargestellt werden. Beispielsweise führt der Befehl *echo (htmlentities("Hallöchen <Freunde>!"));* zur folgenden HTML-Ausgabe: *Hallöchen <Freunde>!.*

Funktion	Bedeutung
$erg=nl2br($str)	wandelt Zeilenumbrüche in * * um
$erg=htmlentities($str)	konvertiert HTML-Zeichen, Umlaute und andere Sonderzeichen, um die Interpretation durch den Internetbrowser zu verhindern
$erg=rawurlencode($str)	konvertiert Umlaute und Sonderzeichen einer Zeichenkette in Prozentzeichen und hexadezimalen ASCII-Wert zur Verwendung in einem URL
$erg=rawurldecode($str)	macht die Konvertierung von *rawurlencode* rückgängig

Tabelle 2.15: Funktionen zur HTML-Bearbeitung von Zeichenketten

Die *rawurl*-Befehle wandeln die eingegebenen Zeichenketten so um, dass sie als Parameter in einem URL übertragen werden können. Dort existiert z. B. das Problem, wie ein Leerzeichen in einem Text dargestellt wird. Eine genormte URL-Darstellung lautet in diesem Fall *%20*. Diese Konvertierungen werden automatisch von *rawurlencode* vorgenommen und von *rawurldecode* wieder auf Empfängerseite rückgängig gemacht.

Datum-/Zeitfunktionen

Genauso wie für die Zeichenketten bietet PHP einige Funktionen zur Bearbeitung von Datums- und Uhrzeitwerten an. Die *checkdate*-Funktion prüft beispielsweise die Eingabe eines Monats, Tages und Jahres auf ein gültiges Datum. Diese Daten können von einer Benutzereingabe stammen.

Funktion	Bedeutung
$erg=checkdate($monat,$tag,$jahr)	überprüft eine Zeitangabe auf Gültigkeit unter Berücksichtung der Schaltjahre und gibt einen Wahrheitswert zurück
$str=date($args)	gibt das Datum in dem durch die Argumente gewünschten Format zurück
$arr=getdate($zeitstempel)	gibt Informationen bezüglich des Datums im UNIX-Zeitstempel als Datenfeld zurück

Tabelle 2.16: Datumsfunktionen

Die *date*-Funktion gibt das aktuelle Datum und die Uhrzeit des Servers zurück. Die Formatierung wird über den Parameter der Funktion anhand der in Tabelle 2.17 abgebildeten Kürzel vorgenommen. So gibt der PHP-Befehl *echo(date("d.m.Y - H:i:s"));* das aktuelle Datum und die Uhrzeit im lesbaren Format *02.06.2009 - 22:54:16* zurück.

Platzhalter	Bedeutung
A	am oder pm
A	AM oder PM
D	Tag des Monats mit führender Null
J	Tag des Monats ohne führende Null
D	abgekürzter Tag
I	vollständig ausgeschriebener Wochentag
F	vollständig ausgeschriebener Monat
M	Monat mit führender Null
M	abgekürzt geschriebener Monat
N	Monat ohne führende Null
H	Stunde im 12-Stunden-Format mit führender Null
H	Stunde im 24-Stunden-Format mit führender Null
G	Stunde im 12-Stunden-Format ohne führende Null
G	Stunde im 24-Stunden-Format ohne führende Null
I	Minuten mit führender Null
S	Sekunden mit führender Null
T	Anzahl der Tage des Monats
W	Wochentag als Zahl; 0 für Sonntag bis 6 für Samstag
Y	zweistellige Jahresangabe
Y	vierstellige Jahresangabe
Z	Tag im Jahr

Tabelle 2.17: Datums- und Zeitformatierung

Zusätzlich existieren Funktionen zum Umgang mit der Uhrzeit, die in Tabelle 2.18 aufgelistet sind. Zunächst wird die Funktion *gettimeofday* vorgestellt, die Zeitinformationen in einem Datenfeld zurück gibt:

```
<html><body>
<?php
  $arr=gettimeofday(); var_dump($arr);
?>
</body></html>
```

Listing 2.56: Test der gettimeofday-Funktion

Die Ausgabe der Funktion lautet *array(4) { ["sec"]=> int(1243976506) ["usec"]=> int(171882) ["minuteswest"]=> int(-120) ["dsttime"]=> int(1) }*. Das assoziative Feld beinhaltet die vergangenen Sekunden und Mikrosekunden seit dem 01.01.1970, 00:00Uhr. Die Variable *minuteswest* beinhaltet die Minuten westlich von der GMT-Zone (Greenwich Mean Time), in diesem Fall bestehen also zwei Stunden Differenz. Die *dsttime* beinhaltet die Korrektur durch Sommerzeit.

Funktion	Bedeutung
$arr=gettimeofday()	gibt die aktuelle Zeit in einem Datenfeld zurück
$erg=microtime()	gibt den aktuellen UNIX-Zeitstempel in Mikrosekunden seit dem 01.01.1970, 00:00Uhr zurück
$erg=time($args)	gibt den aktuellen UNIX-Zeitstempel sekundengenau in der Formatierung der Argumente zurück
$erg=mktime($std,$min,$sek, $monat,$tag,$jahr)	ermittelt den UNIX-Zeitstempel anhand der Zeitangabe

Tabelle 2.18: Zeitfunktionen

Die zweite vorgestellte Funktion ist *microtime* mit µs-Genauigkeit. Diese Funktion eignet sich gut zur Laufzeitmessung von PHP-Quellcode. Der resultierende Zeitstempel nach der Messung der Startzeit wird zerlegt und gespeichert. Aus der Startzeit und der gemessenen Endzeit wird ein neuer Zeitstempel errechnet, der abschließend gerundet wird. So ergibt sich die Zeit in Sekunden. Die gemessene verschachtelte Schleife benötigt im Test ca. 1,4 Sekunden:

```
<html><body>
<?php
  // Start-Zeit
  $messung1=microtime(); $zeittemp=explode(" ",$messung1);
  $messung1=$zeittemp[0]+$zeittemp[1];

  // zu messenger Code
  for ($i=0;$i<10000000;$i++){
    for ($i2=0;$i2<1000;$i2++){
      $i++;
    }
  }

  // Ende-Zeit
  $messung2=microtime(); $zeittemp=explode(" ",$messung2);
  $messung2=$zeittemp[0]+$zeittemp[1]; // Zeitstempel und Nanosek
  $messung=$messung2-$messung1; // Zeit-Differenz
  $messung=substr($messung,0,8); // auf 6 Kommastellen runden
```

Listing 2.57: Eine PHP-Zeitmessung

```
  print('Seitenaufbau in: '.$messung.' Sekunden.');
?>
</body></html>
```

Listing 2.57: Eine PHP-Zeitmessung (Forts.)

Mathematische Funktionen

Neben den Grundrechenoperatoren verfügt PHP zusätzlich über eine Reihe von mathematischen Funktionen, die für Aufgaben des Alltags ausreichen sollten. Tabelle 2.19 zeigt zunächst die wichtigsten trigonometrischen Funktionen, die im Bogenmaß rechnen. Um mit der Zahl Pi zu rechnen, bietet PHP die Funktion *pi()* an. So ergibt *sin(pi()/2)=1*.

Funktion	Bedeutung
$erg=sin($var)	Sinus von *$var*
$erg=cos($var)	Cosinus von *$var*
$erg=tan($var)	Tangens von *$var*
$erg=asin($var)	Arcus-Sinus von *$var*
$erg=acos($var)	Arcus-Cosinus von *$var*
$erg=atan($var)	Arcus-Tangens von *$var*
$erg=atan2($var)	Arcus-Tangens Hyperbolicus von *$var*

Tabelle 2.19: Trigonometrische Funktionen

Die Umwandlung von Grad ins Bogenmaß und umgekehrt bieten die Funktionen *deg2rad* und *rad2deg*. Zusätzlich existiert eine Reihe von Zusatzfunktionen zur Konvertierung von Zahlen in verschiedene Zahlensysteme sowie zum Auf- und Abrunden.

Funktion	Bedeutung
$erg=decbin($var)	konvertiert vom Dezimalsystem ins Binärsystem
$erg=bindec($var)	konvertiert vom Binärsystem ins Dezimalsystem
$erg=dechex($var)	konvertiert vom Dezimalsystem ins Haxadezimalsystem
$erg=hexdec($var)	konvertiert vom Haxadezimalsystem ins Dezimalsystem
$erg=decoct($var)	konvertiert vom Dezimalsystem ins Oktalsystem
$erg=octdec($var)	konvertiert vom Oktalsystem ins Dezimalsystem
$erg=deg2rad($var)	konvertiert Grad zum Bogenmaß
$erg=rad2deg($var)	konvertiert Bogenmaß zu Grad
$erg=base_convert($var,$base1, $base2)	konvertiert zwischen dem Zahlensystem *$base1* in das Zahlensystem *$base2*
$erg=floor($var)	rundet eine Fließkommazahl auf die nächste Ganzzahl ab

Tabelle 2.20: Konvertierungsfunktionen

Funktion	Bedeutung
$erg=ceil($var)	rundet eine Fließkommazahl auf die nächste Ganzzahl auf
$erg=round($var)	rundet einen Wert bei >= x.5 auf und bei < x.5 ab

Tabelle 2.20: Konvertierungsfunktionen (Forts.)

Weitere PHP-Funktionen bieten grundlegende mathematische Berechnungen von Logarithmen, Potenzierung, Absolutwerten und Quadratwurzeln. Mit all diesen Funktionen lassen sich erweiterte Berechnungen zusammensetzen.

Außerdem verfügt PHP über vorgefertigte Funktionen, Minimal- und Maximalwerte aus Listen von Werten zu ermitteln und eine formatierte Ausgabe von Zahlen vorzunehmen.

Funktion	Bedeutung
$erg=abs($var)	Aubsolutwert von *$var*
$erg=exp($var)	Potenz *$var* zur Basis e, der Eulerschen Zahl
$erg=log($var)	natürlicher Algorithmus von *$var*
$erg=log10($var)	natürlicher Algorithmus zur Basis 10
$erg=max($a,$b,$c,...)	größter Wert der Argumente
$erg=min($a,$b,$c,...)	kleinster Wert der Argumente
$erg=number_format($var,$nks, $komma,$tausender)	Formatierung von *$var* in eine Zahl mit Tausender-Trennzeichen, dass in *$tausender* vorgegeben wird; ebenso kann die Anzahl an Nachkommastellen *$nks* vorgegeben werden wie das Trennzeichen selbst in *$komma*
$erg=pow($base,$exp)	potenziert *$exp* zur Basis *$base*
$erg=sqrt($var)	Quadratwurzel von *$var*

Tabelle 2.21: Weitere mathematische Funktionen

Abschließend werden in Tabelle 2.22 Funktionen zur Erzeugung von Zufallszahlen vorgestellt. Mit *getrandmax* können Sie sich die maximale Zufallszahl ermitteln, die sie erzeugen können. So liefert *echo(getrandmax())* die Ausgabe *32767*.

Funktion	Bedeutung
$erg=getrandmax()	ermittelt die höchstmögliche Zahl, die durch die Funktion *rand* erzeugt werden kann
srand($var)	legt über *$var* einen internen Startwert für den Zufallsgenerator fest
$erg=rand($min,$max)	gibt eine Zufallszahl zwischen *$min* und *$max* zurück

Tabelle 2.22: Funktionen für Zufallszahlen

Mit *srand* initialisieren Sie den Zufallsgenerator. Wichtig ist dabei, dass der Parameter von *srand* bereits möglichst zufällig gewählt wird. Ein gleicher Initialwert führt nämlich zu einer gleichen Folge von Zufallszahlen. Mit *rand* erzeugen Sie nun eine Zufallszahl als Ganzzahl in den angegebenen Grenzwerten. Listing 2.58 zeigt die Initialisierung des Zufallszahlengenerators sowie die Erzeugung und Ausgabe von drei Zufallszahlen:

```
<html><body>
<?php
  $zeit=microtime(); $zeitfeld=explode(" ",$zeit);
  $data=$zeitfeld[0]+$zeitfeld[1];
  srand($data); // Initialisierung
  $zufall=rand(0,1000); echo($zufall.'<br>');
  $zufall=rand(0,1000); echo($zufall.'<br>');
  $zufall=rand(0,1000); echo($zufall.'<br>');
?>
</body></html>
```

Listing 2.58: Erzeugung von Zufallszahlen

2.2 Erweiterte Funktionen

Nachdem in Kapitel 2.1 grundlegende Funktionen der Sprache PHP vorgestellt wurden, ist dieses Kapitel auf die Anwendung dieser Funktionalität in typischen kleineren Problemstellungen fokussiert. Die Lösungen dieser Problemstellungen finden oft Verwendung in Anwendungen, bei denen PHP statische HTML-Seiten ergänzt. Dazu gehört Folgendes:

- Auswertung von ausgefüllten HTML-Formularen
- Einführung von Sessions (u. a. zur Realisierung von Warenkörben)
- Weiterleitung auf andere Seiten
- Lesen und Schreiben von Dateien
- Zugriff auf einen FTP-Server zum Dateitransfer
- Zugriff auf eine MySQL-Datenbank
- Automatischer Versand von E-Mails
- Auslesen und Parsen von anderen Homepages

HTML-Formulare auswerten

Ein typischer Anwendungsfall für PHP-Skripte liegt in der Auswertung von ausgefüllten HTML-Formularen, um die Daten des Formulars in eine Datenbank einzutragen. Im ersten Schritt wird ein HTML-Formular benötigt, das für den Testfall aus einem Textfeld, einer Checkbox, einer DropDown-Box und drei verschiedenen Schaltflächen zum Sen-

den der Daten an ein PHP-Formular besteht. Abbildung 2.10 zeigt den Aufbau des HTML-Formulars.

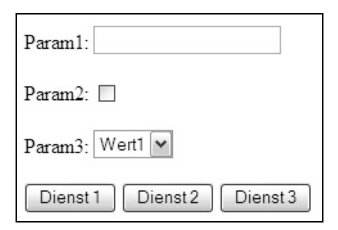


Abbildung 2.10: Ein HTML-Formular mit Steuerelementen

Das Hyptertext-Transfer-Protokoll HTTP erlaubt zwei Methoden, um ausgefüllte Formularelemente von einem Internetbrowser des Clients auf dem Webserver zurückzusenden. Der Quellcode in Listing 2.59 zeigt den Aufbau des HTML-Formulars, das mit der *GET*-Methode zum Webserver zurück gesendet wird. Die Daten werden dabei über den URL (Uniform Resource Locator) im *GET*-Aufruf zum Server gesendet, da dies über das *method*-Attribut des *form*-Tags so angegeben wurde. In dem *action*-Attribut wird angegeben, an welche PHP-Datei das ausgefüllte Formular gesendet werden soll. In diesem Fall handelt es sich um die *fachlogik_get.php*.

Beachten Sie, dass jedes Steuerelement, auch die Schaltflächen, mit einem *name*-Attribut versehen ist. Über die Namen dieser Attribute greift PHP später auf die Formulardaten zu. Nach dem Ausfüllen des Formulars mit Testdaten wurde die Schaltfläche *Dienst1* betätigt. Das erzeugt den folgenden Aufruf auf die *fachlogik_get.php*:

*http://localhost/form/fachlogik_*
get.php?Param1=Frank&Param2=Wert&Param3=Wert1&Dienst1=Dienst+1

Die Parameter werden also in den URL-Aufruf integriert. Dadurch sind Sie in der Lage, den Aufruf über einen Internetbrowser auch zu verändern, indem Sie beispielsweise einen anderen Text hinter *Param1=* im Aufruf platzieren. Sie können den Aufruf also leicht manipulieren, was für Testfälle sinnvoll sein kann. Dadurch ersparen Sie sich das erneute manuelle Ausfüllen des Formulars.

Profitipp

Wenn jemand Ihren Server angreifen will, wird er versuchen, ungültige Daten über ausgefüllte Formulare zu versenden. Wie Sie sehen, ist die Veränderung einer *GET*-Übertragung sehr leicht möglich. Die *POST*-Übertragung, die noch vorgestellt wird, ist nur unwesentlich schwieriger zu manipulieren. Es ist daher unbedingt notwendig, dass Sie alle übergebenen Parameter in PHP nochmals auf Gültigkeit prüfen. Clientseitige Prüfungen sind unzureichend.

Andererseits besitzt die *GET*-Methode auch einige Nachteile. Die Länge der möglichen URLs ist bei einigen Webservern begrenzt, sodass Sie nicht beliebig viele Parameter übergeben können. Zusätzlich existieren besondere Konventionen für Sonderzeichen in einer URL:

```
<html><body>
  <form action="fachlogik_get.php" method="get">
    Param1: <input name="Param1" type="text"><br><br>
    Param2: <input type="Checkbox" name="Param2" value="Wert"><br><br>
    Param3: <select name="Param3" size="1">
      <option>Wert1</option><option>Wert2</option><option>Wert3</option>
    </select><br><br>
    <input name="Dienst1" type="submit" value="Dienst 1">
    <input name="Dienst2" type="submit" value="Dienst 2">
    <input name="Dienst3" type="submit" value="Dienst 3">
  </form>
</body></html>
```

Listing 2.59: Quellcode des GET-Formulars

Listing 2.60 zeigt die serverseitige Auswertung des ausgefüllten Formulars. Der Schlüssel dazu ist das besondere assoziative Datenfeld *$_GET*, das vom PHP-Interpreter automatisch befüllt wird. Die Namen der Felder im assoziativen Array entsprechen den Namen der Steuerelemente im HTML-Formular.

So heißt die Checkbox *Param2*, was dazu führt, dass auch ein Element des Datenfelds *$_GET["Param2"]* in der aufgerufenen PHP-Datei existiert. Dieses Element wird im Beispiel ausgelesen und in der Variablen *$P2* gespeichert. Im Fall der Checkbox ist *$P2=NULL*, wenn die Checkbox nicht angeklickt wurde. Im anderen Fall gilt *$P2="Wert"*, da dieses *value*-Attribut in der HTML-Datei der Checkbox zugeordnet wurde.

Der Inhalt des Textfelds wird aus *$_GET["Param1"]* ausgelesen, während der Inhalt der DropDown-Box in *$_GET["Param3"]* zu finden ist. Der Inhalt von *$_GET["Param3"]* kann *Wert1*, *Wert2* oder *Wert3* sein, je nachdem, welche Option im HTML-Formular gewählt wurde.

Ebenso kann ausgewertet werden, welche *submit*-Schaltfläche zu der *fachlogik_get.php* geführt hat. Die Werte der Schaltflächen werden in *$D1* bis *$D3* abgelegt. Die Variablen *$D2* und *$D3* erhalten jeweils den Wert *NULL*, da die beiden entsprechenden Schaltflächen *Dienst2* und *Dienst3* nicht betätigt worden sind. Die gedrückte Schaltfläche kann mit *$D1="Dienst 1"* ermittelt werden:

```
<html><body>
  <?php
    $P1=$_GET["Param1"]; $P2=$_GET["Param2"]; $P3=$_GET["Param3"];
    $D1=$_GET["Dienst1"]; $D2=$_GET["Dienst2"]; $D3=$_GET["Dienst3"];
    echo (var_dump($P1).'<br>'); echo (var_dump($P2).'<br>');
    echo (var_dump($P3).'<br>');
    echo (var_dump($D1).'<br>'); echo (var_dump($D2).'<br>');
```

Listing 2.60: Quellcode der Auswertung des GET-Formulars fachlogik_get.php

```
    echo (var_dump($D3).'<br>');
  ?>
</body></html>
```

Listing 2.60: Quellcode der Auswertung des GET-Formulars fachlogik_get.php (Forts.)

Die zweite Möglichkeit besteht darin, das ausgefüllte Formular über HTTP-POST an ein PHP-Skript zu übergeben. In diesem Fall werden die Benutzereingaben nicht über den URL, sondern direkt in HTTP-Paketen weitergegeben. Diese Weitergabe vom Client an den Server wird vom HTTP-Protokoll selbst verwaltet und unterliegt im Gegensatz zu der *GET*-Methode keinen Längen- oder Sonderzeichenbeschränkungen. Um ein ausgefülltes Formular per HTTP-POST zu übertragen, müssen Sie lediglich den *form*-Befehl im HTML-Code umändern zu *<form action="fachlogik_post.php" method="post">*. Wie Sie sehen, wird hier eine andere PHP-Datei angesteuert. Die Auswertung eines per *POST* übergebenen Formulars ist zu der *GET*-Übergabe nahezu identisch. Der einzige Unterschied liegt darin, dass bei der Auswertung eines *POST*-Formulars ein anderes assoziatives Datenfeld von PHP ausgewertet werden muss, nämlich *$_POST*. Listing 2.61 zeigt das entsprechende Formular mit der Ausgabe der übegebenen Parameter, wie so oft unter Verwendung des Befehls *var_dump*:

```
<html><body>
  <?php
    $P1=$_POST["Param1"]; $P2=$_POST["Param2"]; $P3=$_POST["Param3"];
    $D1=$_POST["Dienst1"]; $D2=$_POST["Dienst2"]; $D3=$_POST["Dienst3"];
    echo (var_dump($P1).'<br>'); echo (var_dump($P2).'<br>');
    echo (var_dump($P3).'<br>');
    echo (var_dump($D1).'<br>'); echo (var_dump($D2).'<br>');
    echo (var_dump($D3).'<br>');
  ?>
</body></html>
```

Listing 2.61: Quellcode der Auswertung des POST-Formulars fachlogik_post.php

Profitipp

Beachten Sie, dass die Übertragung von ausgefüllten HTML-Formularen mit *GET* oder auch mit *POST* unverschlüsselt erfolgt. Es ist relativ leicht, mit einem Paketanalysator wie WireShark (*http://www.wireshark.org*) im Netzwerk übertragene Daten auszulesen. Bieten Sie dem Benutzer am besten eine geschützte HTTPS-Verbindung an, damit er seine persönlichen Daten eingeben und nicht für andere Personen lesbar übertragen kann.

Sessions und Weiterleitung

Im vorherigen Kapitel wurde gezeigt, wie ein Anwender ein HTML-Formular ausfüllt, das über das HTTP-Protokoll zum Server zurück sendet und wie die Formulardaten über ein PHP-Skript ausgelesen und verarbeitet werden können. Ein Problem besteht

darin, wenn sich der Server diese Formulardaten merken soll, die Daten jedoch nicht so endgültig sind, dass es sich lohnt, sie in einer Datenbank zu speichern. Beispielsweise kann ein Anwender ein großes Formular über mehrere HTML-Seiten eingeben. Oder PHP soll sich merken, ob der Anwender, der sich eben eingeloggt hat, auch wirklich authentifiziert ist. Ein weiterer Anwendungsfall besteht im Aufbau eines Warenkorbs, in den der Benutzer mehrere Artikel hinzufügt, dann zur Kasse navigiert, dort seine Zahlungsweise und Lieferadresse eingibt, um den Bestellvorgang abzuschließen.

Für all diese Fälle wurde ein Sessionmanagement in PHP integriert. Im Gegensatz zu dem zustandslosen HTTP-Protokoll kann sich PHP über eine Session, die dem aktuell geöffneten Internetbrowser des Clients zugeordnet wird, Daten des Anwenders merken. Die Zuordnung erfolgt meist über ein HTTP-Cookie, das vom Webserver an den Client gespeichert und vom Browser gemerkt wird. In diesem Cookie befindet sich ein eindeutiger Identifier, die Session-ID. Erfolgt ein Zugriff von diesem Browser auf eine PHP-Seite, so kann der Webserver die Session-ID vom Browser erfragen und damit auf die temporär gespeicherten Informationen dieses Clients zugreifen. Bei diesen Informationen handelt es sich um Namen und Werte von PHP-Variablen, die der Server in ein spezielles Verzeichnis in einer Datei seines Dateisystems ablegt. Dabei wird der Dateiname identisch zur Session-ID gewählt. In der Konfiguration von PHP wird hinterlegt, wie lange eine Session „leben" kann, wie lange man also diese temporären Informationen zwischenspeichert, bevor sie aufgeräumt werden. Die gesamte Verwaltung der Sessions im Dateisystem erfolgt automatisch durch PHP. Sie als Programmierer müssen sich darum also nicht kümmern.

Eine Session wird mit dem PHP-Befehl *session_start()* initialisiert. Damit wird beim ersten Aufruf eine Session-ID vergeben und eine entsprechende Datei auf dem Server angelegt. Der Zugriff auf die Daten der Session erfolgt – ähnlich wie bei den Daten eines ausgefüllten Formulars – über ein eigenes Datenfeld. Dieses Feld heißt *$_SESSION*. In Listing 2.62 wird eine neue Session gestartet und drei neue leere Variablen *User*, *Pass* und *Auth* in der Session angelegt.

Im Anschluss daran wird ein HTML-Formular mit zwei Textfeldern erstellt und an den Client versendet. Zusätzlich wird die ID der erstellten Session ausgegeben. Die Namen der Variablen im assoziativen Session-Array können, müssen aber nicht, in Hochkommata gesetzt werden:

```
<?php
  session_start();
  $_SESSION[User]=""; $_SESSION[Pass]=""; $_SESSION[Auth]=0;
?>
<html><body>
  <form action="login_server.php" method="post"><pre>
    Benutzer: <input name="frmUser" type="text"><br>
    Kennwort: <input name="frmPass" type="text"><br>
    <input name="Login" type="submit"><br>
  </pre></form>
```

Listing 2.62: Das Login-Formular login.php mit dem ersten Start der Session

```
  <center>
    Sie haben die Session-ID <?php echo session_id()?> vom Server erhalten.
  </center>
</body></html>
```

Listing 2.62: Das Login-Formular login.php mit dem ersten Start der Session (Forts.)

Der Anwender füllt das Formular aus und sendet es an die *login_server.php* zurück. Listing 2.63 skizziert ein Skript, das die eingegebenen Formulardaten prüft und einen Anwender authentifiziert.

Dabei wird zunächst die Session wieder initialisiert, sodass der Zugriff auf die Sessiondaten ermöglicht wird. Zunächst wird geprüft, ob die Variablen *User* und *Pass* in der Session existieren. Das ist dann nicht der Fall, wenn der Anwender die *login_server.php* direkt aufruft, ohne vorher die *login.php* aufgerufen zu haben. In diesem Fall leitet PHP die Ausgabe direkt über das HTTP-Protokoll an die *login.php* weiter, indem der Header des HTTP-Protokolls modifiziert wird.

Profitipp

Wenn Sie bereits HTML-Code an den Client gesendet haben, beispielsweise *<HTML>*, dann können Sie den HTTP-Header nicht mehr modifizieren, da er bereits zum Client gesendet wurde. Die Prüfungen müssen also erfolgen, bevor die erste Ausgabe an den Client erfolgt.

Im Anschluss daran werden die ausgefüllten Formulardaten mit dem Benutzernamen und dem Kennwort ausgelesen. Entspricht der Benutzername der Zeichenkette *frank* und das Passwort der Zeichenkette *geheim*, so ist der Benutzer als Frank authentifiziert und wird in das interne Portal weiter geleitet. Der eingegebene Benutzername wird dabei zunächst mit *strtolower* in Kleinbuchstaben konvertiert und dann mit verglichen. Dadurch ist die Eingabe des Benutzernamens *frank* unabhängig von der Groß- und Kleinschreibung.

Wurde der Benutzername und/oder das Kennwort falsch eingegeben, so erscheint eine Fehlermeldung mit dem Verweis auf die *login.php*.

Der richtige Benutzername und das richtige Kennwort sind in diesem Beispiel fest in PHP codiert. In der Realität würde man nach dem eingegebenen Benutzernamen in einer Datenbank suchen. Wenn er existiert, liest man das richtige Kennwort aus der Datenbank aus und vergleicht es mit dem eingegebenen Kennwort. Sind beide identisch, so ist der Benutzer authentifiziert, in allen anderen Fällen nicht:

```
<?php
  session_start();
  if (!isset($_SESSION[User])||!isset($_SESSION[Pass])){
    header('Location: login.php');
```

Listing 2.63: Die login_server.php prüft den Login und leitet entsprechend weiter

```
  }
  $frmUser=$_POST[frmUser]; $frmPass=$_POST[frmPass];
  if ((strtolower($frmUser)=="frank")&&($frmPass=="geheim")){
    $_SESSION[User]=$frmUser; $_SESSION[Auth]=1;
    header('Location: portal.php');
  }
  else{
    ?>
      <html><body>
        <center>
          Ihre Eingabe war leider falsch!<br>
          <a href="login.php">Zurück um Login...</a>
        </center>
      </body></html>
    <?php
  }
?>
```

Listing 2.63: Die login_server.php prüft den Login und leitet entsprechend weiter (Forts.)

Listing 2.64 beschreibt die Datei *portal.php*. Dort muss man zunächst prüfen, ob serverseitig eine Authentifizierungsvariable existiert und falls ja, ob diese den Wert 1 hat. Nur in diesem Fall handelt es sich um einen authentifizierten Anwender; in den anderen Fällen erfolgt eine Weiterleitung zu der Loginmaske. Eine Umgehung der Loginmaske ist – selbst wenn ein Angreifer Kenntnis von der Existenz der Datei *portal.php* hat – nicht möglich. Wenn alles in Ordnung ist, erfolgt eine personalisierte Willkommensnachricht, da der Name des Anwenders in der Session gespeichert wurde:

```
<?php
  session_start();
  if (!isset($_SESSION[Auth])){
    header('Location: login.php');
  }
  if ($_SESSION[Auth]!=1){
    header('Location: portal.php');
  }
?>
<html><body>
  <h1>Hallo <?php echo $_SESSION[User]?>, Herzlich Willkommen im Portal!</h1>
</body></html>
```

Listing 2.64: Beim erfolgreichen Login gelangt man auf die portal.php

Zum Abschluss dieses Kapitels werden in Tabelle 2.23 die wichtigsten Befehle zur Verwaltung von Sessions vorgestellt. Interessant ist dabei, dass in einer Session registrierte Variablen auch wieder gelöscht werden können.

Funktion	Bedeutung
session_start()	startet eine neue Session oder übernimmt eine bereits vorhandene
$erg=session_id()	gibt den eindeutigen Identifier der eigenen, aktuellen Session zurück
$erg=session_encode()	liefert eine Liste mit allen abgespeicherten Variablen der aktuellen Session
session_unregister("name")	entfernt eine Variable mit dem Namen *name* aus der Liste der Variablen einer Session
session_unset()	löscht den Inhalt aller Variablen, die in der aktuellen Session deklariert wurden; die Session selbst bleibt jedoch bestehen
session_destroy()	beendet eine Session und löscht alle Daten, die in der Session verwendet wurden; die Session-ID wird wieder freigegeben

Tabelle 2.23: Session-Befehle von PHP

Das @ zur Fehlerunterdrückung

In den folgenden Kapiteln wird der Zugriff auf das Dateisystem des Servers, der Upload von Dateien auf einem FTP-Server und der Zugriff auf eine Datenbank beschrieben. Man kann sich leicht vorstellen, dass bei diesen Funktionen eine Vielzahl von Fehlern möglich ist: Es fehlen Schreibrechte auf eine Datei, das FTP-Kennwort ist falsch oder der Datenbankserver ist offline.

Die Befehle, die diese Funktionen realisieren, geben im Fehlerfall meist direkt eine PHP-Fehlermeldung zum Internetbrowser des Clients zurück. So liefert der Befehl *$datei=fopen("counter.txt","w");* beispielsweise die folgende Fehlermeldung, wenn das PHP-Skript keinen Schreibzugriff auf die zu schreibende Datei besitzt (eine genauere Betrachtung der Funktionen zum Schreiben von Dateien erfolgt im nächsten Kapitel):

Warning: fopen(counter.txt) [function.fopen]: failed to open stream: Permission denied in C:\EigeneDateien\HTTP\counter.php on line 7

Ein Kunde, der auf die Seite zugreift, soll jedoch nicht von dieser Meldung abgeschreckt werden. Unmittelbar vor jede Meldung, die einen Fehler oder eine Warnmeldung ausgeben kann, können Sie ein @ platzieren, um auftretende Meldungen zu unterdrücken. Wichtig ist dabei, dass das @ unmittelbar für die fehleranfällige Funktion geschrieben wird und nicht vor die ganze Zeile. Der korrekte Befehl zur Unterdrückung der Meldung lautet also: *$datei=@fopen("counter.txt","w");*.

Schlägt der Befehl *fopen* fehl, wird jetzt keine Fehlermeldung mehr ausgegeben. Dennoch müssen Sie dafür sorgen, dass das Programm korrekt ausgeführt wird. Dazu müssen Sie wissen, dass dieser Befehl im Fehlerfall die Variable *$datei* als Wahrheitswert ausgibt und diesen auf *FALSE* setzt. Sie müssen also direkt hinter dem Befehl im Quellcode eine Prüfung mit *if($datei===FALSE){...}* vornehmen, um die korrekte Ausführung des Skripts zu gewährleisten.

Lesen und Schreiben von Dateien

In diesem Kapitel wird vorgestellt, wie Sie Dateien im Dateisystem des Webservers auslesen und schreiben können. Das macht dann Sinn, wenn sich eine komplexe Datenbankanbindung nicht lohnt. Als Beispiel wird im Folgenden eine typische Anwendung eines Zählers der Besucher auf eine Homepage erstellt. Dieser einzelne Wert müsste sonst in einer eigenen Datenbanktabelle abgelegt werden.

Dabei wird zunächst versucht, eine Datei mit dem Namen *counter.txt* aus demselben Verzeichnis, in dem das Skript ausgeführt wird, auszulesen. Diese Datei ist natürlich beim ersten Aufruf noch nicht vorhanden, sodass der *fopen*-Befehl *FALSE* zurückgibt. Wenn die Datei existiert, wird in der Variablen *$datei* in Listing 2.65 eine Referenz auf die offene Datei abgelegt. Diese Referenz nennt man auch Resource oder Handle.

Wenn die Datei existiert, liest der Befehl *fgets* bis zum nächsten Zeilenumbruch, bis zum Ende der Datei oder 10 Zeichen (in dieser Reihenfolge) aus. Die ausgelesenen Daten werden dann in *$counter* gespeichert. In dem Fall, dass die Datei noch nicht existierte, ist *$counter=""*. Dann setzt PHP den Wert auf 0. In jedem Fall wird der Zähler erhöht und die geöffnete Datei wieder geschlossen.

Danach wird die Datei wieder geöffnet, diesmal mit Schreibzugriff. Mit dem Befehl *fwrite* wird der erhöhte Wert des Counters in der Datei serverseitig gespeichert und die Datei abschließend wieder geschlossen. Im HTML-Teil des Skripts wird dann nur noch der Zählerstand ausgegeben:

```
<?php
  $datei = @fopen("counter.txt","r+");
  $counter = @fgets($datei, 10);
  if($counter == "") $counter = 0;
  $counter++;
  @fclose($datei);
  $datei = @fopen("counter.txt","w");
  @fwrite($datei, $counter);
  @fclose($datei);
?>
<html><body>
  <?php echo $counter?> Mal wurde bereits Ihre Seite angezeigt.
</body></html>
```

Listing 2.65: Ein einfacher PHP-Counter

Beim Testen des Skripts ist zunächst erfreulich, dass es korrekt funktioniert. Es gibt bei dieser Art des Dateizugriffs jedoch einige Probleme.

Zunächst muss man darauf hinweisen, dass der Zähler bei jedem Reload der Seite hochgezählt wird. In der Realität handelt es sich jedoch um denselben Besucher. Abhilfe schafft hier die Einrichtung einer Session, wie es bereits im vorletzten Kapitel vorgestellt wurde. Nur beim Start der Session wird der Wert einmalig aus der Datei gelesen, erhöht

und wieder gespeichert. Ist der Wert bereits in der Session vorhanden, muss er bei einem Reload der Seite lediglich aus der Session geholt werden.

Ein weiteres Problem kann bei der Zugriffsberechtigung im Dateisystem des Servers auftreten. Wenn die Datei noch nicht existiert, versucht der *fwrite*-Befehl, sie anzulegen. Es kann jedoch sein, dass das PHP-Skript nicht die Berechtigung hat, in das Dateisystem des Servers zu schreiben. In diesem Fall schlägt der Schreibzugriff fehl. Ein Lösungsansatz kann darin bestehen, die Textdatei zunächst clientseitig anzulegen, auf dem Server per FTP (File Transfer Protocol) hochzuladen und die Zugriffsrechte per FTP zu erhöhen.

Ein weiteres Problem tritt auf, wenn mehrere Benutzer gleichzeitig auf diese Datei zugreifen. Das Skript läuft vereinfacht in dieser Reihenfolge ab:

1. Lesender Dateizugriff, um den Counter-Wert zu holen.
2. Counter-Wert erhöhen.
3. Erhöhten Counter-Wert schreiben.

Nehmen wir an, dass zwei Benutzer fast gleichzeitig auf das Skript zugreifen. Benutzer A führt Schritt 1 aus. Dann führt Benutzer B den Schritt 1 aus. Beide haben dann denselben Counter-Wert, beispielsweise 23 und erhöhen ihn um 1. Beide Benutzer schreiben danach die Zahl 24 in die Datei, obwohl der Zähler eigentlich um zwei Werte erhöht werden sollte.

Profitipp

Seien Sie immer vorsichtig, wenn mehrere Benutzer per PHP auf eine einzelne Ressource zugreifen können. Dadurch können so genannte „Concurrency-Probleme" entstehen. Für den Dateizugriff bietet PHP die Funktion *flock* an, um den Zugriff auf eine Datei exklusiv zu sperren. Durch einen Lock der Datei vor Schritt 1 und ein Unlock nach Schritt 3 wird sichergestellt, dass die Schritte 1 bis 3 ohne Unterbrechung ausgeführt werden und kein zweiter Benutzer den Ablauf unterbrechen kann. Ein zweiter Benutzer dieses Skripts würde dann seinerseits bei dem Lock-Versuch etwas warten, bis der erste Benutzer die Ressource wieder freigibt.

Zum Abschluss dieses Kapitels werden in Tabelle 2.24 die wichtigsten PHP-Funktionen für den Zugriff auf das Dateisystem des Servers vorgestellt und kurz erläutert. PHP verfügt über eine große Vielzahl von Befehlen zum Zugriff auf das Dateisystem und die Verzeichnisstruktur des Servers, die Sie in aktuellen Onlinequellen wie *http://www.selfphp.de/funktionsreferenz/dateisystem_funktionen/* nachlesen können.

Funktion	Bedeutung
$fh=fopen($var,$op)	öffnet die in *$var* angegebene Datei oder den URL und gibt eine Referenz auf das geöffnete Objekt zurück; mögliche Werte für die Operation *$op* sind: "a": öffnen zum Schreiben; Referenz zeigt auf das Ende der Datei; eine nicht existierende Datei wird angelegt "a+": öffnen zum Lesen und Schreiben; Referenz zeigt auf das Ende der Datei; eine nicht existierende Datei wird angelegt "r": öffnen zum Lesen; Referenz zeigt auf den Anfang der Datei "r+": öffnen zum Lesen und Schreiben; Referenz zeigt auf den Anfang der Datei "w": öffnen zum Schreiben; Referenz zeigt auf den Anfang der Datei; existierende Datei wird auf 0 Byte gesetzt; eine nicht existierende Datei wird angelegt "w+": öffnen zum Lesen und Schreiben; Referenz zeigt auf den Anfang der Datei; eine nicht existierende Datei wird angelegt
$erg=fclose($fh)	schließt eine zuvor mit *fopen* geöffnete Datei; bei Erfolg wird *TRUE*, sonst *FALSE* zurückgeliefert
$erg=fgets($fh,$var)	liest Daten aus der Dateireferenz *$fh* ein; entweder bis Zeilenumbruch, Dateiende oder bis zur Anzahl an Zeichen, die in *$var* angegeben wurde
$erg=fgetcsv($fh,$var,$trenner)	liest eine Zeile aus der geöffneten CSV-Datei (Comma Separated Values) *$fh* aus; der Parameter *$var* beinhaltet die Anzahl der zu lesenden Zeichen und muss größer sein als die längste Zeile in der Datei, da sonst das Ende der Zeile nicht gefunden wird; in *$trenner* wird das Trennzeichen der CSV-Datei angegeben; die Rückgabe ist ein Datenfeld
fwrite($fh,$var)	schreibt die als Zeichenkette in *$var* übergebenen Daten in die Datei *$fh*
$erg=is_file($var)	wenn die Datei existiert und es eine reguläre Datei ist, gibt *is_file TRUE*, sonst *FALSE* zurück; in *$var* wird der Pfad und der Name der Datei als Zeichenkette übergeben
$erg=file_exists($var)	überprüft, ob eine in *$var* übergebene Pfad- und Dateiangabe existiert und gibt *TRUE* zurück, wenn das der Fall ist, und ansonsten *FALSE*
$erg=filectime($var)	gibt das Datum und die Uhrzeit der letzten Änderung einer Datei in *$var* als UNIX-Zeitstempel zurück
$erg=filesize($var)	gibt die Größe der Datei, die ggf. zusammen mit Pfadangabe in *$var* angegeben wird, zurück; bei einem Zugriffsfehler wird *FALSE* zurückgegeben

Tabelle 2.24: PHP-Befehle zum Zugriff auf das Dateisystem des Webservers

Funktion	Bedeutung
$erg=flock($fh,$op)	schützt eine Datei *$fh* vor Operationen, die in *$op* übergeben werden; *$op* kann sein: *LOCK_SH*: Verriegelung für Lesezugriff *LOCK_EX*: exklusive Verriegelung für Schreibzugriffe *LOCK_UN*: gibt eine Verriegelung wieder frei *LOCK_NB*: verhindert, dass die Funktion während der Verriegelung blockiert; diese Konstante kann zusätzlich zu den anderen Konstanten angegeben werden
$erg=unlink($var)	löscht die in *$var* übergebene Datei und gibt *FALSE* zurück, wenn die angegebene Datei nicht gelöscht werden konnte

Tabelle 2.24: PHP-Befehle zum Zugriff auf das Dateisystem des Webservers (Forts.)

Als besondere Funktion ist *fgetcsv* zu betonen, die eine Zeile aus einer CSV-Datei ausliest und als Datenfeld im Ergebnis zurück liefert. Das CSV-Format wird von einer Vielzahl von Anwendungen wie SAP und Microsoft Excel zum Datenaustausch angeboten. Mithilfe dieser Funktion können Sie also unter anderem einen Datenimport aus einer Fremdanwendung realisieren.

FTP-Funktionen

Eine übliche Methode, um Dateien von einem Client zu einem Server zu übertragen, ist die Verwendung des FTP-Protokolls (File Transfer Protocol). Während über HTTP als Anwendungsprotokoll statischer HTML-Code und HTML-Formulare übertragen werden, ist FTP für den Transfer von Dateien zuständig.

Das Hochladen auf den FTP-Server und das Herunterladen von Dateien vom FTP-Server zum Client funktioniert ähnlich wie das im letzten Kapitel vorgestellte Lesen und Schreiben von Dateien aus PHP heraus; Sie müssen Folgendes tun:

1. eine Verbindung zum FTP-Server herstellen
2. eine oder mehrere Dateien hoch- und/oder herunterladen
3. abschließend die geöffnete Verbindung wieder schließen

Im Folgenden soll beispielhaft eine einzelne Datei vom Client zum Server hochgeladen werden. Sie brauchen die folgenden Parameter, um einen solchen Transfer durchzuführen:

- die Adresse des FTP-Servers
- den FTP-Benutzernamen, mit dem Sie sich am FTP-Server anmelden
- das FTP-Kennwort, mit dem Sie sich am FTP-Server anmelden
- den Pfad und den Namen der hochzuladenden Datei auf dem Client
- den Pfad und den Namen der Datei, wie sie auf dem Server gespeichert werden soll

Diese Parameter können alle oder teilweise in einem HTML-Formular eingegeben werden. Wenn Sie nicht alle Parameter eingeben wollen, können Sie im PHP-Skript auch

feste Werte vergeben. Wenn Sie beispielsweise den Benutzernamen und das Kennwort als Konstanten im PHP-Code festlegen, authentifiziert sich lediglich das PHP-Skript gegen den FTP-Server. Jeder Anwender, der das Skript ausführen kann, kann somit auch Dateien hochladen. Listing 2.66 skizziert das PHP-Skript für einen FTP-Zugriff. In diesem Beispiel sind alle Parameter direkt im PHP-Code gesetzt und können vom Benutzer nicht geändert werden:

```
<?php
  // Notwendige Parameter für den FTP-Zugriff
  $server="212.227.89.9"; $user="benutzername"; $pass="kennwort";
  $quelle="C:/test.txt"; $ziel="/httpdocs/test.txt";
  $fh=@ftp_connect($server);
  $login=@ftp_login($fh,$user,$pass);
?>
<html><body>
  <?php
  if ((!$fh)||(!$login)) {
    ?>
    <h1>Ftp-Verbindung nicht hergestellt!<h1>
    <p>Verbindung mit dem Server als Benutzer <?php echo $user?> nicht möglich!
    <?php
    die;
  }
  else {
    ?><p>Sie sind verbunden mit dem Server als Benutzer <?php echo $user?>.</p><?php
    $upload=@ftp_put($fh,$ziel,$quelle,FTP_ASCII);
    if (!$upload) {
      ?><p>Upload fehlgeschlagen!</p><?php
    }
    else {
      ?><p>Datei <?php echo $quelle?> erfolgreich geschrieben.</p><?php
    }
    @ftp_quit($fh);
    ?>Verbindung wurde wieder getrennt</p><?php
  }
?>
</body></html>
```

Listing 2.66: Ein FTP-Upload über PHP

Mit dem Befehl *ftp_connect* verbinden Sie sich unter Angabe der IP-Adresse auf TCP/IP-Ebene mit dem Server. Auf diese Verbindung können Sie wie auf eine geöffnete Datei mit einem Handler *$fh* zugreifen. Um die Zugriffsrechte zu ermitteln, müssen Sie sich nun mit *ftp_login* unter Angabe des Benutzernamens und des Kennworts authentifizieren.

Wenn die Verbindung und/oder der Login nicht erfolgreich waren, können Sie nichts hochladen und das Skript bricht ab. Mit *ftp_put* wird nun versucht, die Datei von der Festplatte des Clients in das Dateisystem des Servers hochzuladen. Dabei müssen Sie in der folgenden Konstante angeben, ob Sie diese Datei als Text *FTP_ASCII* oder binär *FTP_BINARY* hochladen wollen. Anhand der Rückgabe der PUT-Funktion können Sie ermitteln, ob das Hochladen erfolgreich war. In allen Fällen wird die Verbindung zum Server nach dem Hochladeversuch wieder geschlossen.

Wenn alle Parameter korrekt angegeben wurden, erhalten Sie folgende Ausgabe:

Sie sind verbunden mit dem Server als Benutzer benutzername.

Datei C:/test.txt erfolgreich geschrieben :-)

Verbindung wurde wieder getrennt

Hinweis

Eine gute Übung besteht darin, diesem PHP-Skript ein HTML-Formular vorzuschalten, damit alle Parameter vom Anwender eingegeben werden können.

Beachten Sie bitte, dass die maximale Ausführungszeit eines Skripts oft über den Webserver eingeschränkt wird. In der Konfigurationsdatei *php.ini* ist in der aktuellen XAMPP-Version über den Parameter *max_execution_time=60* die Ausführungszeit eines PHP-Skripts auf 60 Sekunden beschränkt. Wenn Sie große Dateien hochladen und die Uploadgeschwindigkeit nicht sonderlich hoch ist, wie es bei DSL üblich ist, kann das Skript durchaus länger als 60 Sekunden ausgeführt werden. Ist das der Fall, wird die Ausführung durch den PHP-Interpreter unmittelbar beendet. Die Verbindung wird dann auch nicht wieder korrekt geschlossen.

In Tabelle 2.25 werden die wichtigsten PHP-Befehle des FTP-Protokolls aufgelistet und kurz beschrieben.

Funktion	Bedeutung
$fh=ftp_connect($server);	verbindet sich mit dem angegebenen Server und gibt eine Referenz auf die Verbindung zurück
$erg=ftp_login($fh,$user,$pass)	loggt einen Benutzer mit seinem Kennwort auf einer existierenden Verbindung ein; das Ergebnis ist ein Wahrheitswert über den Erfolg der Anmeldung
$erg=ftp_put($fh,$ziel,$quelle,$art)	lädt die in *$quelle* angegebene Datei zum FTP-Server in *$ziel* hoch; die Art des Transfers erfolgt als Text (*FTP_ASCII*) oder binär (*FTP_BINARY*); der Erfolg des Transfers wird als Wahrheitswert zurückgegeben
$erg=ftp_get($fh,$ziel,$quelle,$art)	lädt die in *$quelle* angegebene Datei vom FTP-Server lokal in *$ziel* herunter; die Art des Transfers erfolgt als Text (*FTP_ASCII*) oder binär (*FTP_BINARY*); der Erfolg des Transfers wird als Wahrheitswert zurückgegeben

Tabelle 2.25: PHP-Befehle zum FTP-Protokoll

Funktion	Bedeutung
$erg=ftp_cdup($fh)	wechselt auf dem Server in das Hauptverzeichnis; der Erfolg wird als Wahrheitswert zurückgegeben
$erg=ftp_chdir($fh,$dir)	wechselt auf dem Server in das angegebene Verzeichnis *$dir*; der Erfolg wird als Wahrheitswert zurückgegeben
$erg=ftp_mkdir($fh,$dir)	erzeugt auf dem Server das Verzeichnis *$dir*; der Erfolg wird als Wahrheitswert zurückgegeben
$arr=ftp_nlist($fh,$dir)	gibt die Dateien und Unterverzeichnisse von *$dir* als Datenfeld von Zeichenketten zurück
$erg=ftp_rename($fh,$neu,$alt)	benennt eine Datei auf dem FTP-Server von *$alt* in *$neu* um; der Erfolg wird als Wahrheitswert zurückgegeben
$erg=ftp_mdtm($fh,$datei)	gibt den UNIX-Zeitstempel der letzten Änderung von *$datei* zurück
$erg=ftp_delete($fh,$datei)	löscht *$datei* vom FTP-Server; der Erfolg wird als Wahrheitswert zurückgegeben
$erg=ftp_close($fh)	schließt eine geöffnete FTP-Verbindung; der Erfolg wird als Wahrheitswert zurückgegeben

Tabelle 2.25: PHP-Befehle zum FTP-Protokoll (Forts.)

Zugriff auf eine MySQL-Datenbank

Einer der wichtigsten Anwendungsfälle von PHP ist der server-seitige Zugriff auf eine Datenbank, mit deren Inhalten dynamisch HTML-Tabellen aufgebaut (beispielsweise zu kaufende Artikel) und Eingaben von Benutzern als Daten in einer anderen Tabelle abgelegt werden können. Das können unter anderem Kundendaten oder Bestellungen sein. Die Kombination der Skriptsprache PHP mit der leicht administrierbaren, internettauglichen Open-Source-Datenbank MySQL (*http://www.mysql.org*) hat seit dem Jahr 2000 zu der erheblichen Verbreitung und damit zum Siegeszug von PHP beigetragen.

Diese Kombination ermöglicht den Aufbau einer so genannten 3-Tier-Architektur (auch: 3-Schichten-Architektur), die im Gegensatz zu statischen Webseiten eine datenbankabhängige Gestaltung von Inhalten erlaubt. Der Client bildet dabei das Frontend mit der Präsentationsschicht, das auch als GUI (Graphical User Interface) bezeichnet wird. Im Frontend können HTML, JavaScript, Java Applets und/oder Flash-Animationen zum Einsatz kommen. Die Anwendungsschicht, die oft auch als Fachlogik oder Businesslogik bezeichnet wird, bildet die „Intelligenz der Anwendung" und wird von den PHP-Skripten auf dem Webserver realisiert. Die dritte Schicht wird als Datenschicht, Datenzugriffsschicht oder Persistenzschicht bezeichnet. Hier werden abrufbare Daten organisiert abgelegt und es können neue Daten durch Eingabe in der Präsentationsschicht und Prüfung bzw. Aufbereitung in der Anwendungsschicht hinzugefügt werden. Die Datenschicht wird meist durch eine relationale Datenbank aufgebaut, die aus Datenbanktabellen besteht und die über die Sprache SQL (Structured Query Language) von der Anwendungsschicht aus angesprochen wird.

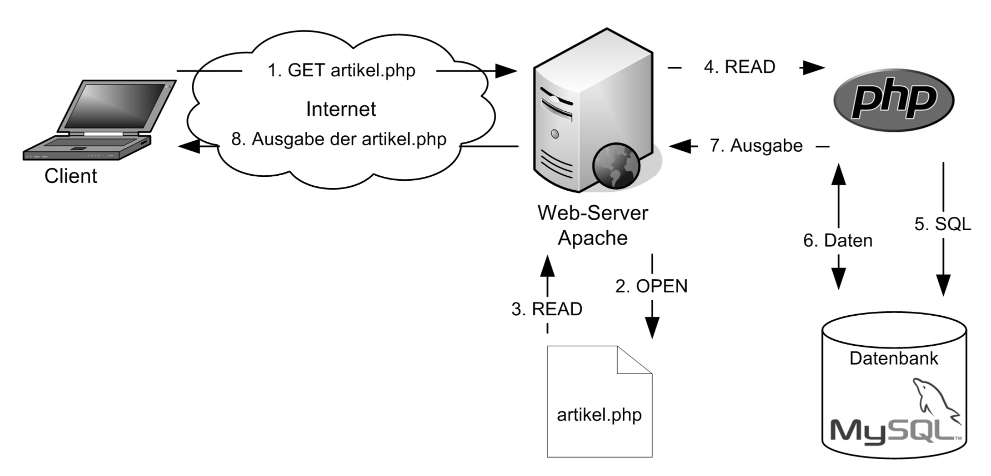


Abbildung 2.11: 3-Schichten-Architektur: Client, PHP und MySQL

Der Aufruf erfolgt, indem zunächst wie gewohnt vom Internetbrowser des Clients eine Anfrage auf eine PHP-Datei des Webservers über das HTTP-Protokoll abgesetzt wird. Der Webserver liest diese Datei dann aus seinem Dateisystem aus und übergibt sie an den PHP-Interpreter. Ähnlich wie Befehle zum Zugriff auf das Dateisystem des Servers oder zum Öffnen einer FTP-Verbindung zum Datenaustausch, bietet PHP einen Befehlssatz zum Zugriff auf eine MySQL-Datenbank an. Dabei können SQL-Befehle abgesetzt werden, die lesenden oder schreibenden Zugriff auf die Datenbank ermöglichen. Ein *SELECT*-Kommando holt beispielsweise Daten aus der Datenbank ab und speichert diese Daten in einer zweidimensionalen Datenstruktur, also ähnlich wie in einer Tabelle, für PHP ab.

Das PHP-Skript durchläuft dann diese Datenstruktur und erzeugt auf dieser Basis eine dynamische HTML-Antwort, zum Beispiel als HTML-Tabelle. Die Ausgabe des PHP-Skripts wird dann über den Webserver als Antwort der Anfrage zum Client zurückgesendet.

Um einen solchen Zugriff zu ermöglichen, müssen zunächst Tabellen in der Datenbank vorhanden sein. Das installierte XAMPP-Paket beinhaltet neben dem Webserver Apache und der Skriptsprache PHP auch eine Installation der Datenbank MySQL sowie ein bekanntes Tool zur Administration der Datenbank. Dieses Tool wird *phpMyAdmin* genannt und ist selbst in PHP programmiert worden.

Starten Sie über das XAMPP Control Panel zunächst Apache und MySQL und rufen dann *http://localhost/xampp/* auf. Im Hauptmenü auf der linken Seite finden Sie unter *Tools* einen Link auf *phpMyAdmin*. Alternativ dazu können Sie auch direkt *http://localhost/phpmyadmin/* aufrufen. Über HTML-Formulare können Sie nun eine neue Datenbank anlegen. Als Beispielanwendung wird eine Datenbank mit dem Namen *boerse* angelegt.

Innerhalb der Datenbank können Sie nun Tabellen anlegen. Im Beispiel wird die Tabelle *ag* angelegt, in der die Namen der Aktiengesellschaften hinterlegt sind. Zusätzlich existiert eine Tabelle *kurse*, in der die Kurse der letzten 30 Tage für jede Aktiengesellschaft festgehalten wird. Die Datenbank mit ihren beiden Tabellen wird in Abbildung 2.12 im *phpMyAdmin*-Tool dargestellt.

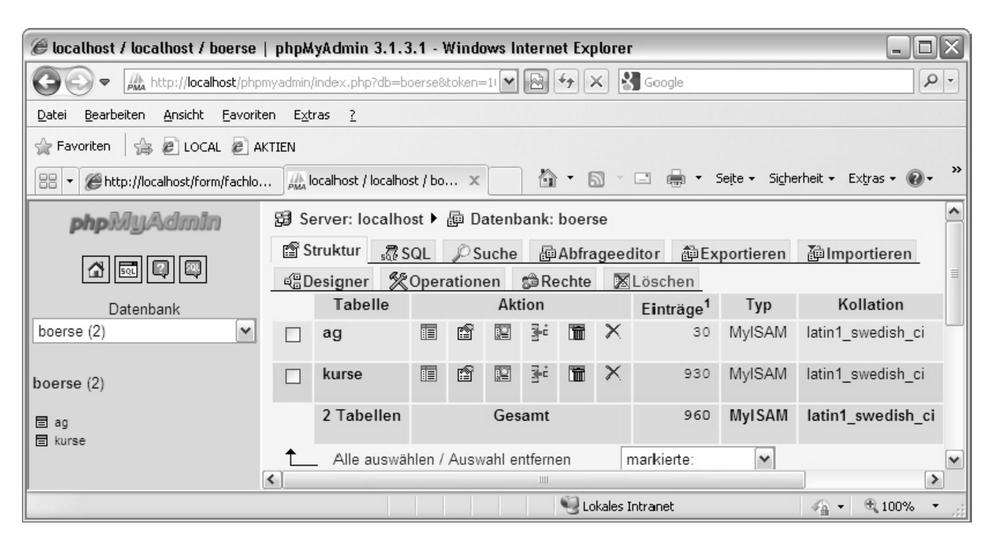


Abbildung 2.12: Struktur der Börsendatenbank

Ein Grundsatz der Datenbankmodellierung besagt, dass innerhalb einer Tabelle keine Datensätze doppelt auftreten sollen. Um jeden Datensatz eindeutig zugreifbar zu machen, verwendet man eindeutige Identifier für jeden Datensatz. Diese Identifier nennt man Primärschlüssel. Aus diesem Grund beinhaltet die Tabelle *ag* zwei Felder, nämlich die eindeutige ID und den Namen der Aktiengesellschaft. Zusätzlich wird angegeben, dass sowohl das Feld *ID* als auch das Feld *name* in der Datenbank nicht *NULL* sein darf. Beide Felder müssen also stets gefüllt sein. Die Struktur der Tabelle *ag* ist in Abbildung 2.13 skizziert.

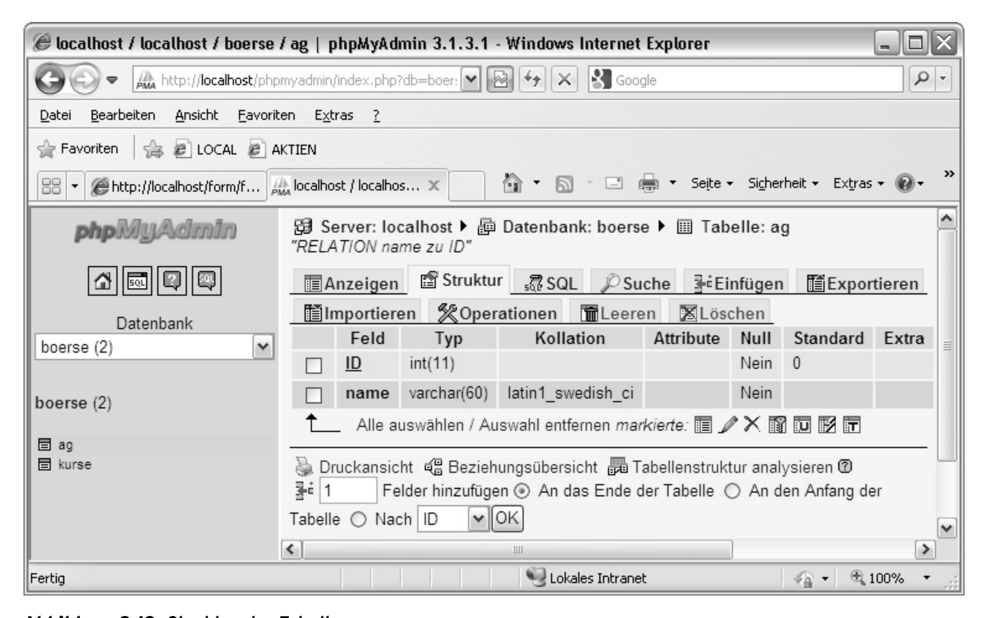


Abbildung 2.13: Struktur der Tabelle ag

Auffallend ist noch der Typ jedes Felds. Wie auch eine Programmiersprache besitzt eine Datenbank eine Reihe von Datentypen, die man bei der Erstellung der Tabelle angeben muss. Die Tabelle 2.26 zeigt eine Übersicht der MySQL-Datentypen sowie den benötigten Speicherplatz für jeden Eintrag.

Datentyp	Speicherplatz	Beschreibung
TINYINT	1 Byte	Ganzzahlen von 0 bis 255 oder von -128 bis +127
SMALLINT	2 Byte	Ganzzahlen von 0 bis 65 535 oder von -32 768 bis +32 767
MEDIUMINT	3 Byte	Ganzzahlen von 0 bis 16 777 215 oder von -8 388 608 bis +8 388 607.
INT	4 Byte	Ganzzahlen von 0 bis ~4,3 Millionen oder von -2 147 483 648 bis +2 147 483 647
BIGINT	8 Byte	Ganzzahlen von 0 bis 18 446 744 073 709 551 615 oder von -9 223 372 036 854 775 808 bis 9 223 372 036 854 775 807
FLOAT	4 Byte	vorzeichenbehaftete Fließkommazahl zur Darstellung annähernder numerischer Datenwerte; Wertebereich von -3,402823466^38 bis -1,175494351^38, 0 und 1,175494351^38 bis 3,402823466^38
DOUBLE	8 Byte	vorzeichenbehaftete Fließkommazahl zur Darstellung annähernder numerischer Datenwerte; Wertebereich von ca. -1,798^308 bis ca. -2,225^-308, 0 und ca. 2,225^-308 bis ca. 1,798^308
DECIMAL	abh. von der maximalen Anzahl der eingegebenen Stellen	vorzeichenbehaftete Fließkommazahl zur Speicherung exakter numerischer Datenwerte
DATE	3 Byte	Datumsangabe im Format "YYYY-MM-DD"; Wertebereich von 01.01.1000 bis 31.12.9999
DATETIME	8 Byte	Datumsangabe im Format "YYYY-MM-DD hh:mm:ss"; der Wertebereich entspricht DATE
TIMESTAMP	4 Byte	Zeitstempel; Wertebereich von 01.01.1970 bis 2037
TIME	3 Byte	Zeit zwischen -838:59:59 und +839:59:59; Ausgabe im Format "hh:mm:ss"
YEAR	1 Byte	Jahr zwischen 1901 bis 2155 bei zweistelliger und zwischen 1970 bis 2069 bei vierstelliger Speicherung
CHAR	abh. von der maximalen Anzahl der eingegebenen Zeichen	Zeichenkette fester Länge, wobei jedes Zeichen einen Wertebereich von 0 bis 255 besitzt (ANSII)

Tabelle 2.26: MySQL-Datentypen

Datentyp	Speicherplatz	Beschreibung
VARCHAR	abh. von der String-Länge	Zeichenkette variabler Länge, wobei jedes Zeichen einen Wertebereich von 0 bis 255 besitzt (ANSII)
BLOB	abh. von den eingegebenen Daten	binäres Objekt mit variablen Daten; weitere Typen sind *TINYBLOB*, *MEDIUMBLOB* und *LONGBLOB*
TEXT	abh. von der String-Länge	wie *BLOB*; berücksichtigt jedoch beim Sortieren und Vergleichen die Groß- und Kleinschreibung; weitere Typen sind: *TINYTEXT*, *MEDIUMTEXT*, *LONGTEXT*
ENUM	1 oder 2 Byte	Liste von Werten; max. 65 535 eineindeutige Elemente möglich
SET	abh. von den eingegebenen Daten (1 bis 8 Byte)	String-Objekt mit verschiedenen Variablen; max. 64 Mitglieder sind möglich

Tabelle 2.26: MySQL-Datentypen (Forts.)

Über *phpMyAdmin* kann die Tabelle mit Daten befüllt, geändert oder auch gelöscht werden. Dadurch ergibt sich die in Abbildung 2.14 skizzierte Darstellung.

			ID	name
☐	✎	✕	1	ADIDAS-SALOMON AG
☐	✎	✕	2	ALLIANZ AG VNA O.N
☐	✎	✕	3	ALTANA AG O.N.
☐	✎	✕	4	BASF AG O.N.
☐	✎	✕	5	BMW
☐	✎	✕	6	Bayer

Abbildung 2.14: Inhalt der Tabelle ag

Die zweite Tabelle wird als *kurse* bezeichnet und enthält Kurssaten zu den angegebenen Aktiengesellschaften. Neben einer eigenen ID als Primärschlüssel (Kapitel 3.1.3). Zusätzlich werden der Tag, der Wert des Kurses und die ID der Aktiengesellschaft abgelegt. Wenn in einer Tabelle der Primärschlüssel einer anderen Tabelle eingebunden wird, wird dieser Schlüssel als Fremdschlüssel bezeichnet.

Es stellt sich die Frage, wieso zwei Tabellen angelegt werden, wobei eine Tabelle lediglich aus den Namen der Aktiengesellschaft besteht. Wieso wird der Name der Aktiengesellschaft nicht anstelle des Fremdschlüssels eingebunden? Gibt es Regeln zur Erstellung von Datenfeldern?

Die Anwendung der Regeln zur Erstellung von Datenbanktabellen wird als ER-Modellierung (Entity Relationship) bezeichnet. Dazu gehört die Anwendung der so genannten Normalisierung. Um eine Datenbank bereits in die erste Normalform zu bringen, muss jedes Datenfeld aus einem atomaren Wert bestehen, den man nicht weiter zerlegen kann. So muss der Name einer Person in den Feldern *Vor-* und *Nachname* abgelegt werden. Das Gleiche gilt für Adressen, die in *PLZ*, *Ort*, *Straße* und *Hausnummer* abgelegt werden müssen. Dadurch werden Suchen und statistische Auswertungen der Daten ermöglicht.

Nun zu der Frage, warum überhaupt zwei Tabellen notwendig sind und nicht die Namen der Aktiengesellschaften direkt in die Tabelle *kurse* geschrieben werden. Abbildung 2.15 zeigt die Struktur dieser Tabelle.

Nehmen wir an, dass eine Aktiengesellschaft umbenannt wird, was in der heutigen Zeit nicht unüblich ist. Bei 1000 Einträgen in die Kurstabelle müssten dann 1000 Zeichenketten anpassen; bei einer separaten Tabelle nur eine einzige Zeichenkette. Das ist jedoch der weniger wichtige Grund. Wenn man davon ausgeht, dass nach 400 Änderungen der Datenbankserver abstürzt, befindet sich die gesamte Tabelle in einem ungültigen Zustand. Man kann nicht mehr sagen, welche Daten bereits geändert wurden und welche nicht. Oft fällt der Fehler erst wesentlich später auf. Man spricht in diesem Zusammenhang von einer „Updateanomalie". Um diese zu vermeiden, dürfen in Datenbanken keine Daten redundant abgelegt werden. Das wird durch höhere Normalformen erreicht.

Wenn mehrere Operationen auf einer Datenbank entweder vollständig ausgeführt werden müssen oder durch einen Fehler während der Ausführung gar nicht ausgeführt werden dürfen, spricht man von Transaktionen. Transaktionsmanagement wird von neueren Versionen der MySQL-Datenbank ebenso unterstützt wie von professionellen Datenbanksystemen wie Oracle oder MS SQL Server.

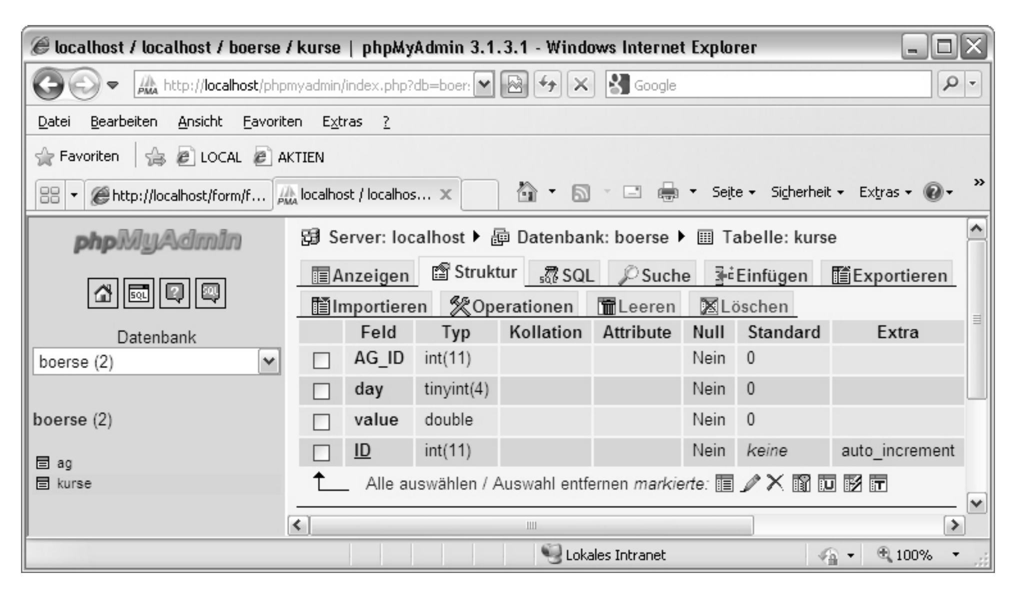


Abbildung 2.15: Struktur der Tabelle kurse

Zu kritisieren an der Datenstruktur ist der Datentyp *double* für die Kurse, da dieser Datentyp den Wert nur annähernd repräsentiert. Natürlich ist die Genauigkeit eines *double*-Wertes wesentlich genauer als die benötigten zwei Nachkommastellen für Währungen. Probleme kann es jedoch geben, wenn man mit Währungen rechnet, die Rundung von Werten in Kauf nimmt und Werte auf Gleichheit überprüft. Als Regel kann man festhalten, dass für alle Währungen stets der Datentyp *decimal* verwendet werden sollte, der eine exakte numerische Darstellung bietet.

Hinweis

Legen Sie als Übung die Datenbank mit den beiden Tabellen in der dargestellten Struktur über *phpMyAdmin an*. Wenn Sie das getan haben, befüllen Sie die Tabellen über den Menüpunkt *Einfügen* in *phpMyAdmin* mit Testdaten. Mithilfe der folgenden PHP-Skripte können Sie dann auf diese Daten zugreifen.

Abbildung 2.16 zeigt die mit Testdaten gefüllte Tabelle *kurse*.

AG_ID	day	value	ID
1	1	153.04	1
1	2	131.61	2
1	3	117.14	3
1	4	106.59	4
1	5	109.79	5
1	6	90.03	6
1	7	81.93	7
1	8	63.9	8

Abbildung 2.16: Tabelle kurse

Der Zugriff auf eine Datenbank ist mit dem Zugriff auf eine Datei oder auf einen FTP-Server zu vergleichen. Er verläuft grundsätzlich in drei Schritten:

1. Verbindung zur Datenbank aufbauen
2. Interaktion mit der Datenbank durchführen
3. Verbindung mit der Datenbank abbauen

Die Dienste des gesamten Datenbankzugriffs werden in eigene Funktionen gekapselt. Dadurch können die Funktionen wie PHP-eigene Befehle verwendet werden. Alle Funktionen werden im Beispiel in einer einzigen Datei, der *DBzugriff.inc.php* gespeichert. Diese Datei wird von den Skripten hinzugefügt, die Datenbankfunktionalität verwenden.

Das Aufbauen der Verbindung in der Funktion *DB_open()* erfolgt in zwei Schritten. Im ersten Schritt verbinden Sie das Skript mit dem Datenbankserver. Dabei müssen Sie die IP-Adresse des Servers (bei XAMPP können Sie für lokale Tests *localhost* verwenden), Ihren Benutzernamen und Ihr Kennwort eingeben, mit denen sich das Skript gegen den Server authentifiziert. In MySQL können verschiedene Benutzer verschiedene Zugriffsrechte auf eine Datenbank besitzen.

Der Zugriff auf den Server wird dann mit dem Befehl *mysql_connect* ausgeführt. Im zweiten Schritt verbinden Sie sich dann mit einer Datenbank auf dem Server unter Verwendung des Befehls *mysql_select_db*, in unserem Beispiel mit der Datenbank *boerse*. Wenn beides erfolgreich ist, gibt die eigene Funktion den Wert *TRUE*, ansonsten *FALSE* zurück.

Das Schließen der Verbindung zum Datenbankserver soll mit der Funktion *DB_close()* erfolgen. Diese verwendet lediglich den Befehl, der eine offene Verbindung wieder schließt.

Profitipp

In diesem Buch wird als Benutzer stets *root* ohne Kennwort vergeben. Das ist aus Gründen der Sicherheit natürlich nicht akzeptabel und darf lediglich zu Testzwecken verwendet werden. Mit diesen Benutzerrechten können bei einer manipulierten Eingabe in das Skript ganze Tabellen gelöscht werden, unter anderem auch die Tabelle der möglichen Benutzer des MySQL-Servers. Auf diese Weise kann also der gesamte Datenbankserver lahmgelegt werden. Oder ein Angreifer kann Zugriff auf persönliche Daten nehmen, die nach dem Datenschutzgesetz nicht zugreifbar sein dürften. Die Verwendung von Administratorrechten für diese Zugriffe stellt dann eine fahrlässige Handlung dar, bei der Sie als Programmierer unter Umständen haftbar gemacht werden können.

Abschließend kapselt der erste Teil der *DBzugriff.inc.php* die Funktion *DB_error* die MySQL-Funktion *mysql_error*, bei der detailliertere Angaben über die letzte Fehlermeldung beim Datenbankzugriff ausgegeben werden können. Sie fragen sich vielleicht, aus welchem Grund eine einzelne Funktion in einer anderen Funktion mit allgemeinerem Namen verpackt wird. Der Grund dafür liegt darin, dass alle Funktionen, die einen Bezug zum MySQL-Server haben, ausschließlich in einer einzelnen Datei abgelegt sein sollen. Das bildet die Datenzugriffsschicht. Ihre Anwendung verwendet dann diese Datei, um wiederum Funktionen der Fachlogik verwenden zu können:

```
<?php
function DB_open(){
  $DB_Host="localhost";
  $DB_Benutzername="root"; $DB_Passwort=""; $DB_Name="boerse";
  $OK=@mysql_connect($DB_Host,$DB_Benutzername,$DB_Passwort);
  if (!$OK)
    return FALSE;
  else{
    if (@mysql_select_db($DB_Name)==1){
      return TRUE;
    }
    else{
      return FALSE;
    }
  }
}
function DB_close(){
  @mysql_close();
}
function DB_error(){
  return @mysql_error();
}
```

Listing 2.67: Einzubindende Datei DBzugriff.inc.php, erster Teil

Die für die Fachlogik interessanten Funktionen des Datenzugriffs werden im zweiten Teil der *DBzugriff.inc.php* realisiert. Dabei werden drei Dienste angeboten, die Daten aus der Datenbank auslesen:

- *DB_AGs()* liest alle Aktiengesellschaften aus der Datenbank und gibt die Namen als Datenfeld von Zeichenketten zurück.
- *DB_MW($AG)* liefert den Mittelwert aller Börsenkurse einer Aktiengesellschaft, deren Name als Parameter übergeben wird.
- *DB_Kurs($AG,$tag)* gibt einen einzelnen Aktienkurs der einzugebenden Aktiengesellschaft an einem bestimmten Tag aus.

Alle drei Dienste verwenden den PHP-Befehl *mysql_query*, der eine Zeichenkette als Parameter erhält. Diese Zeichenkette enthält einen SQL-Abfragebefehl, der mit *SELECT* beginnt. Im Anschluss daran werden die Spalten aus der Datenbank angegeben, die man in der Ausgabe verwenden möchte. Das teilweise eingesetzte Schlüsselwort *DISTINCT* sorgt dafür, dass keine Datensätze im Ergebnis doppelt vorhanden sind. Der *FROM*-Teil eines SQL-Befehls gibt den Namen der Datenbanktabelle an, aus der man Daten auslesen will. Durch Angabe eines *WHERE*-Teils kann das Ergebnis eingeschränkt werden. So gibt der Befehl *SELECT ID FROM ag WHERE name='".$AG."'"* nur die ID einer Aktiengesellschaft zurück, deren Name gleich dem Namen des Parameters *$AG* ist. Da die Namen als Zeichenkette abgespeichert werden, muss der Parameter im SQL-Befehl in Hochkommata gesetzt werden. Mit dem Zusatz *ORDER BY* sortieren Sie das Ergebnis nach einer oder mehreren Spalten, im Fall der Funktion *DB_AGs()* nach der *ID*, und zwar in aufsteigender Reihenfolge. Die aufsteigende Reihenfolge wird mit *ASC* (engl.: ascending) festgelegt. Eine absteigende Reihenfolge könnten Sie mit *DESC* (engl.: descending) angeben.

Die Funktion *DB_MW($AG)* zeigt, dass Sie mehrere SQL-Befehle schachteln können. Mit dem inneren *SELECT*-Befehl holen Sie sich die ID zu einem gegebenen Namen einer Aktiengesellschaft. Diese ID wird als Einschränkung in der *WHERE*-Klauses beim Zugriff auf eine andere Datenbanktabelle verwendet, da Sie ja nur den Mittelwert einer Aktiengesellschaft mit dieser angegebenen ID berechnet haben wollen. Grundlegende Funktionen wie eine Addition oder eine Mittelwertberechnung kann der Datenbankserver selbst durchführen. MySQL bietet hier die Funktion *avg(value)* an, die den Mittelwert direkt als Ergebnis ausgibt.

In allen Fällen befindet sich das Ergebnis der SQL-Abfrage in einer lokalen Variablen namens *$data*. Das Ergebnis einer SQL-Abfrage wird als Resultset bezeichnet. Nun müssen Sie dieses Ergebnis auswerten. Dazu bietet PHP einige Befehle an.

Mit *mysql_fetch_array* wird eine Zeile nach der anderen als Datenfeld zurückgegeben. Mit der Angabe von *MYSQL_ASSOC* wird ein assoziatives Feld erzeugt. Aus der Datenbanktabelle *ag* mit *ID=5* und *name=BMW* ergeben sich die Feldelemente *$datensatz[ID]=5* und *$datensatz[name]="BMW"*. Dieses Datenfeld kann dann von der Fachlogik weiter verarbeitet werden.

Auch mit dem Befehl *mysql_fetch_row* holen Sie eine Zeile aus der Ergebnistabelle. Bei der Mittelwertberechnung ist lediglich ein einziger Wert in der Ergebnistabelle. Die Zeile wird über *mysql_fetch_row* ermittelt und liefert ein numerisches Datenfeld mit einem ein-

zigen Element. Dieses Element wird abschließend über *return $datensatz[0];* zurückgegeben.

Ein weiterer, häufig verwendeter Befehl ist *mysql_num_rows*, der die Anzahl der Datensätze in der Ergebnistabelle zurückgibt. Diese Zahl kann als Zähler für Schleifen oder als Indiz für die Anzahl der ermittelten Datensätze verwendet werden:

```
function DB_AGs(){
  $data=@mysql_query("SELECT ID,name FROM ag ORDER BY ID ASC");
  if ($data==FALSE) return FALSE;
  $data_ausgabe=Array();
  while ($datensatz=@mysql_fetch_array($data,MYSQL_ASSOC)){
    $data_ausgabe[]=$datensatz;
  }
  return $data_ausgabe;
}
function DB_MW($AG){
  $data=@mysql_query("SELECT avg(value) FROM kurse WHERE
       AG_ID = (". "SELECT DISTINCT ID FROM ag WHERE name='".$AG."'".")");
  if ($data==FALSE) return FALSE;
  $datensatz=@mysql_fetch_row($data);
  return $datensatz[0];
}
function DB_Kurs($AG,$tag){
  // AG_ID holen
  $data=@mysql_query("SELECT DISTINCT ID FROM ag WHERE name='".$AG."'");
  if ($data==FALSE) return FALSE;
  if (@mysql_num_rows($data)!=1) return FALSE;
  $datensatz=@mysql_fetch_row($data);
  $AG_ID=$datensatz[0];
  // Börsenwert holen
  $data=@mysql_query("SELECT DISTINCT value FROM kurse
                                                   WHERE AG_ID=".$AG_ID);
  if ($data==FALSE) return FALSE;
  $datensatz=@mysql_fetch_row($data);
  return $datensatz[0];
}
?>
```

Listing 2.68: Einzubindende Datei DBzugriff.inc.php, zweiter Teil

Im nächsten Schritt werden kurz die typischen SQL-Befehle vorgestellt, die bei einem Datenbankzugriff verwendet werden. Wenn Sie den SQL-Befehlssatz ausführlich lernen wollen, empfehlen sich Onlinequellen. Im Referenzhandbuch zu MySQL ist beispiels-

weise unter *http://dev.mysql.com/doc/refman/5.1/de/select.html* die formale Syntax der *SELECT*-Anweisung beschrieben. Daran lässt sich nachvollziehen, dass allein zur Sprache SQL eigene Bücher verfasst werden können. Bedenken Sie dabei, dass es sich dabei um eine eigene Sprache handelt. Der Vorteil ist, dass SQL unhabhängig von der verwendeten Programmiersprache und sogar relativ unabhängig von der verwendeten Datenbank ist. Wenn Sie also einmal den SQL-Befehlssatz beherrschen, sind sie auch für andere Programmiersprachen und Datenbanken gut gerüstet.

Befehl	Beschreibung
CREATE DATABASE shop;	legt eine neue Datenbank auf dem Server an
CREATE TABLE kunde (ID bigint(20) NOT NULL auto_increment, nachname varchar(30) NOT NULL, vorname varchar(30) NOT NULL, P RIMARY KEY (ID));	legt eine neue Tabelle an, wobei zuvor eine Datenbank ausgewählt werden muss; in diesem Beispiel werden zwei Felder mit Datentyp *varchar* und 30 Zeichen angelegt, die befüllt werden müssen; zusätlich existiert eine ID, die automatisch vergeben wird und die den Primärschlüssel der Tabelle darstellt
DROP DATABASE alt;	löscht eine existierende Datenbank mitsamt aller Tabellen und Daten
INSERT INTO kunde (nachname,vorname) VALUES ('Dopatka','Frank');	fügt einen neuen Datensatz zu einer existierenden Tabelle hinzu; dabei müssen die Namen der Spalten angegeben werden, die zu befüllen sind sowie die Daten, die den neuen Datensatz bilden
UPDATE kunde SET nachname='Maier' WHERE nachname='Dopatka'	ändert/aktualisiert Datensätze in einer Tabelle, die dem Kriterium in der *WHERE*-Klausel entsprechen
SELECT * FROM kunde WHERE vorname='Frank' LIMIT 20	liest Spalten aus der angegebenen Tabelle aus (*entspricht allen Spalten) und beschränkt die Ausgabe durch Einträge, die der *WHERE*-Klausel entsprechen (hier: nur Leute mit dem Vornamen *Frank* zurückgeben; zusätzlich kann noch ein Limit angegeben werden (hier: max. 20 Einträge zurückgeben)

Tabelle 2.27: Typische SQL-Befehle

Ein weiteres wichtiges Merkmal für Datenbanken ist die Möglichkeit, Transaktionen durchzuführen. Dabei wird eine Reihe von Zugriffen auf die Datenbank entweder ganz oder gar nicht ausgeführt. Das ist sinnvoll, wenn mehrere Benutzer gleichzeitig Änderungen an einem einzigen Datenstamm durchführen können.

Im folgenden Beispiel wird der Ausschnitt eines Quellcodes skizziert, der eine Kontobuchung tätigt. Dabei soll ein Betrag von 100 Euro umgebucht werden. Das soll aber nur möglich sein, wenn das Quellkonto noch positives Guthaben besitzt.

Das Problem liegt darin, dass generell bei mehreren hintereinander programmierten Datenbankabfragen nicht garantiert werden kann, dass diese ohne Unterbrechung auch hintereinander ausgeführt werden. Ein anderes Skript kann stets zwischen zwei Zugriffen auf die Datenbank den Datenstamm manipulieren. So kann ein Betrag mehrfach von einem Konto abgebucht werden mit der Wirkung, dass jedes Skript für sich zwar lokal gesehen korrekt funktioniert, der Kontostand letztlich jedoch negativ ist.

Der Ausschnitt in Listing 2.69 zeigt, wie Sie mehrere SQL-Abfragen zu einer Transaktion bündeln. Dazu müssen Sie die SQL-Befehle *START TRANSACTION* und *BEGIN* absetzen. Alle folgenden Kommandos an die Datenbank können nicht durch andere Zugriffe auf dieselben Daten unterbrochen werden. Die Folge der Kommandos wird entweder mit *COMMIT* vollständig oder mit *ROLLBACK* gar nicht ausgeführt. So verhindern Sie, dass Ihr Datenstamm inkonsistent wird. Damit Transaktionen funktionieren, müssen Sie Tabellen vom Typ *InnoDB* oder *BDB* verwenden. Der standardmäßig eingestellte Tabellentyp *MyISAM* unterstützt bei MySQL keine Transaktionen. Dafür sind die Zugriffe auf diese Tabellen jedoch schneller, da kein aufwendiges Transaktionsmanagement berücksichtigt werden muss:

```
mysql_query("START TRANSACTION");
mysql_query("BEGIN");
mysql_query("UPDATE konto SET stand=stand-100 WHERE nummer=4711");
mysql_query("UPDATE konto SET stand=stand+100 WHERE nummer=4712");
// das SELECT wird hier auf Daten angewendet,
// die sich noch nicht "endgültig" in der Datenbank befinden:
$res=mysql_query("SELECT stand FROM konto WHERE nummer=4711");
$stand=mysql_result($res,0,0);
if ($stand<0){
  mysql_query("ROLLBACK");
}
else{
  // erst jetzt werden die Daten wirklich geschrieben!
  mysql_query("COMMIT");
}
```

Listing 2.69: Ausschnitt aus einem PHP-Quellcode mit Transaktionen

Zusätzlich zu den SQL-Befehlen, die als Zeichenketten auf den Datenbankserver abgesetzt werden, sollten Sie die gängigen PHP-Befehle zum Umgang mit MySQL-Datenbankverbindungen kennen. Diese werden in Tabelle 2.28 kurz vorgestellt.

Befehl	Beschreibung
$erg=mysql_connect ($host,$user,$pass)	verbindet sich mit einem MySQL-Server unter Angabe von Benutzername und Kennwort; der Erfolg wird als Wahrheitswert zurückgegeben
mysql_close($db)	schließt eine geöffnete Verbindung zu einem MySQL-Server
$erg=mysql_error()	liefert den Fehlertext des zuvor ausgeführten MySQL-Befehls
$erg=mysql_errno()	liefert die Fehlernummer des zuvor ausgeführten MySQL-Befehls
$res=mysql_list_dbs()	gibt die Datenbanken des MySQL-Servers als Resultset zurück; das Resultset kann u. a. über *$data=mysql_fetch_array($res)* zeilenweise zurückgegeben und mit *echo $data[Database]* ausgegeben werden

Tabelle 2.28: PHP-Befehle für den Zugriff auf eine MySQL-Datenbank

Befehl	Beschreibung
$res=mysql_list_tables($db)	dibt die Namen der Tabellen aus der Datenbank *$db* als Resultset zurück, *$db* wird als Zeichenkette übergeben
mysql_select_db($db)	wählt eine Datenbank, deren Name als Zeichenkette in *$db* übergeben wurde, zur weiteren Verwendung aus
$erg=mysql_query($var)	setzt eine SQL-Abfrage auf den Datenbankserver ab; das Ergebnis ist je nach SQL-Statement ein Resultset (z. B. bei einer lesenden *SELECT*-Anweisung) oder ein Wahrheitswert (z. B. bei einer schreibenden *UPDATE*-Anweisung)
$arr=mysql_fetch_array ($res,$var)	liest eine Zeile aus einem Resultset *$res* als Datenfeld aus; mit *$var* kann der Aufbau des Felds gewählt werden: *$var=MYSQL_ASSOC* bildet ein assoziatives Feld *$var=MYSQL_NUM:* bildet ein nummerisches Feld
$arr=mysql_fetch_row($res)	liest eine Zeile aus einem Resultset *$res* als nummerisch indiziertes Datenfeld
$erg=mysql_num_rows($res)	liefert die Anzahl der Einträge in einem Resultset

Tabelle 2.28: PHP-Befehle für den Zugriff auf eine MySQL-Datenbank (Forts.)

Die Datenzugriffsschicht der Beispielanwendung ist in der Datei *DBzugriff.inc.php* realisiert und diskutiert worden. Im zweiten Schritt muss nun ein HTML-Frontend erstellt werden, über das Sie die Dienste aufrufen können. Dieses Formular besteht aus zwei Textfeldern zur Eingabe einer Aktiengesellschaft sowie eines Tages und zusätzlich aus drei Schaltflächen, diee die drei Dienste repräsentieren, die von der Datenzugriffsschicht bereitgestellt werden, nämlich

- das Auslesen aller Aktiengesellschaften (kein Eingabeparameter notwendig)
- das Anzeigen eines Kurses einer gegebenen Aktiengesellschaft (zwei Eingabeparameter notwendig)
- das Berechnen des Kursmittelwerts einer gegebenen Aktiengesellschaft (ein Eingabeparameter notwendig)

Alle drei Schaltflächen leiten die eingegebenen Daten über HTTP-POST an die Datei *fachlogik.php* weiter:

```
<html>
  <head><title></title></head>
  <body>
  <form action="fachlogik.php" method="post"><pre>
    AG: <input name="frmAG" type="text" value="BMW"><br>
    Tag: <input name="frmTag" type="text" value="5"><br>
    <input name="funcAGs" type="submit" value="alle AGs ausgeben"
                                              style="width: 11em"><br>
    <input name="funcKurs" type="submit" value="Kurs anzeigen"
```

Listing 2.70: Das Eingabeformular

```
                                                style="width: 11em"><br>
    <input name="funcMW" type="submit" value="MW berechnen"
                                                    style="width: 11em">
  </pre></form>
  </body>
</html>
```

Listing 2.70: Das Eingabeformular (Forts.)

Dadurch entsteht ein Frontend, das in Abbildung 2.17 dargestellt wird.

Abbildung 2.17: HTML-Formular zum Auslesen der Börsendaten aus der Datenbank

Die Datei *fachlogik.php* bildet das Bindeglied zwischen der Präsentationsschicht und der Datenzugriffsschicht. In ihrem ersten Teil, der in Listing 2.71 dargestellt wird, werden die Eingaben aus dem ausgefüllten HTML-Formular entgegengenommen und in PHP-Variablen abgelegt. Hier können in einer realen Anwendung auch Prüfungen der Gültigkeit von Eingaben vorgenommen werden. Im Anschluss daran werden die Funktionen eingebunden, die die Dienste der Datenzugriffsschicht beinhalten:

```
<?php
  $frmAG=$_POST["frmAG"]; $frmTag=$_POST["frmTag"];
  $funcAGs=$_POST["funcAGs"]; $funcKurs=$_POST["funcKurs"];
  $funcMW=$_POST["funcMW"];
  if ($funcAGs!=NULL){
    $func=1;
  }
  else if ($funcKurs!=NULL){
    $func=2;
  }
  else if ($funcMW!=NULL){
    $func=3;
  }
  else{
    $func=0;
  }
```

Listing 2.71: Fachlogik, Teil 1: Auswerten der übergebenen Formulardaten

```
  require("DBzugriff.inc.php");
?>
```

Listing 2.71: Fachlogik, Teil 1: Auswerten der übergebenen Formulardaten (Forts.)

Der zweite Teil der Fachlogik besteht aus der Ausgabe der Antwort auf die HTTP-Anfrage. Nach dem Öffnen der Verbindung zur Datenbank wird durch die *switch*-Anweisung untersucht, welchen Dienst der Benutzer angefordert hat. Dementsprechend werden die Daten aus der Datenbank angefordert, die als Rückgabewerte der Funktionen aus der Datenzugriffsschicht festgehalten werden. Im Anschluss daran erfolgt die HTML-Aufbereitung dieser Daten für die Ausgabe. Abschließend wird die Verbindung zur Datenbank wieder geschlossen:

```
<html>
  <head><title>Meine Börse.</title></head>
  <body>
  <?php
  if (!DB_open()){
    echo "Fehler beim oeffnen der DB-Verbindung: ".DB_error();
  }
    else{
      switch ($func){
        case 1: // alle AGs
          $AGs=DB_AGs();
          foreach ($AGs as $index => $datensatz){
            echo ('Die DAX-AG mit der ID '.$datensatz[ID].' ist
                                            '.$datensatz[name].'<br>');
          }
        break;
        case 2: // Kurs eines Tages
          $Kurs=DB_Kurs($frmAG,$frmTag);
            echo ('Der Kurs von '.$frmAG.' am Tag '.$frmTag.' war
                                     '.number_format($Kurs,2).'EUR.');
          break;
        case 3: // Mittelwert
          $MW=DB_MW($frmAG);
          echo ('Der Mittelwert des Kurses von '.$frmAG.' ist
                                        '.number_format($MW,2).'EUR.');
          break;
        default:
          echo ('Ungültiger Dienst-Aufruf!');
          break;
      }
      DB_close();
    }
```

Listing 2.72: Fachlogik, Teil 2: Ausführen der Dienste

```
  ?>
<br><a href="gui.html">zurück...</a>
</body></html>
```

Listing 2.72: Fachlogik, Teil 2: Ausführen der Dienste (Forts.)

Versand von E-Mails

In diesem Kapitel wird beschrieben, wie Sie mit PHP E-Mails selbst versenden können. Dazu ist nur ein einziger Befehl vorhanden: *mail($empfanger,$betreff,$nachricht,$sender).* Dieser Befehl sendet die in der Variablen *$nachricht* gespeicherte Nachricht von dem Sender *$sender* an den Empfänger *$empfänger*. Zusätzlich wird noch der Betreff in der Variablen *$betreff* übergeben. Es fällt vielleicht auf, dass die Absendeadresse beliebig eingegeben werden darf. Das E-Mail-Protokoll verlangt nämlich keine Authentifizierung vom Absender.

Zusätzlich stellt sich die Frage, warum dem E-Mail-Versand ein eigenes Kapitel gewidmet wird. Die Ursache liegt nämlich darin, dass der *mail*-Befehl unter einem XAMPP-Server auf einem Windows-Betriebssystem nicht funktioniert und *FALSE* zurückgibt. Das liegt daran, dass kein Mail-Server existiert, an den die zu sendende E-Mail übergeben werden kann. Linux bietet hier bereits eigene Lösungen an.

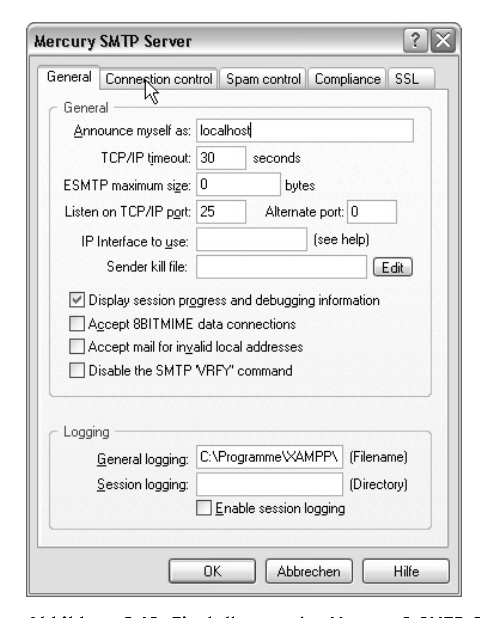


Abbildung 2.18: Einstellungen des MercuryS-SMTP-Server

In dem XAMPP-Paket ist der Mail-Server Mercury integriert, der für Windows-Plattformen noch konfiguriert werden muss und der für lokale Tests ausreichend ist. Starten Sie im ersten Schritt den Mail-Server in dem XAMPP Control Panel. Wenn der Mail-Server im Status *Running* ist, klicken Sie im Control Panel auf *Admin*. Daraufhin öffnet sich das Fenster *Mercury/32* zur Administration des Mail-Servers. Klicken Sie nun im Hauptmenü auf *Configuration* und dann auf *MercuryS SMTP Server*. Dort müssen Sie im Writer *General* im Feld *Announce myself as* den Wert *localhost* eintragen.

Im Register *Connection Control* können Sie das Häkchen *Do not permit SMTP relaying of non-local mail* wegnehmen. So können Sie auch Zieladressen von anderen Domains lokal testen. Mit Klick auf *OK* können Sie dann das Fenster schließen.

Klicken Sie im Anschluss daran nochmals auf *Configuration* und dann auf *MercuryS SMTP Client*. Hier müssen Sie unter *Indentify myself as* nochmals *localhost* und als *Name server* die IP-Adresse *127.0.0.1*. Diesen Dialog beenden Sie mit *Save*.

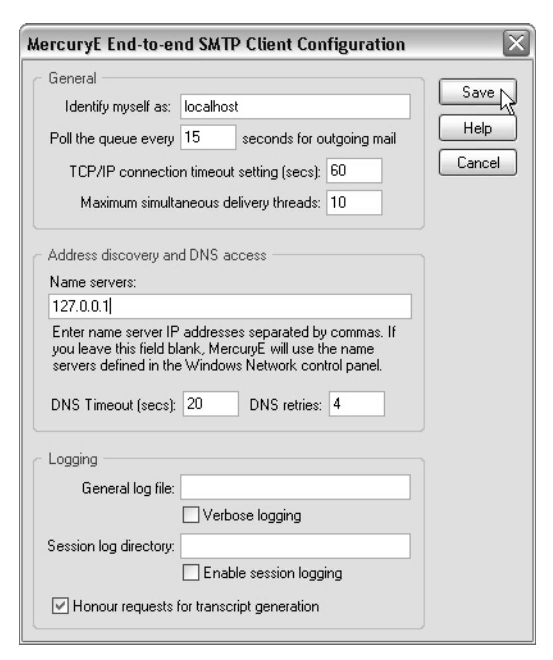


Abbildung 2.19: Einstellungen des MercuryS SMTP Client

Im nächsten Schritt müssen Sie noch den Benutzer einrichten, also ein Mail-Konto auf dem Server anlegen. Klicken Sie dazu auf *Configuration*, dann auf *Manage local users* und abschließend auf *add*. Es wird der Benutzer *FrankDopatka* mit dem Passwort *test* angelegt. Die Administrationsrechte sind nicht zwingend notwendig. Mit *OK* beenden Sie den Dialog und mit *Close* schließen Sie die Benutzerverwaltung.



Abbildung 2.20: Einrichtung eines neuen Benutzers auf dem SMTP-Server

Der Mail-Server ist nun betriebsbereit. Um E-Mails von dem Server abzuholen, müssen Sie nun noch ein Mail-Client-Programm konfigurieren. Ein erfolgreicher Test wurde mit MS Outlook Express 6 durchgeführt. Aber auch andere Mail-Clients sollten problemlos konfiguriert werden können. Um Verbindung mit dem Mail-Server aufnehmen zu können, benötigen Sie folgende Angaben:

E-Mail-Adresse:	FrankDopatka@localhost
Posteingangsserver POP3:	127.0.0.1
Postausgangsserver SMTP:	127.0.0.1
Benutzername:	FrankDopatka
Kennwort:	test

Mit diesen Einstellungen können Sie nun das in Listing 2.73 dargestellte Mail-Skript ausführen. Der PHP-Interpreter erzeugt damit eine neue E-Mail und gibt sie an den Mail-Server weiter:

```
<html><body>
<h1>Sende Test-Mail...</h1><br><br>
<?php
  $Mailnachricht="Dies ist eine Test-Mail";
  $Empfaenger = "FrankDopatka@localhost";
  $Mailbetreff = "Test-Mail";
  if(!@mail($Empfaenger, $Mailbetreff, $Mailnachricht,
                                        "from:FrankDopatka@localhost"))
  {
    ?>Beim Versand der e-Mail ist ein Fehler aufgetreten!<br><?php
  }
  else{
    ?>E-Mail erfolgreich versendet.<?php
  }
?>
</body></html>
```

Listing 2.73: PHP-Skript zum Mail-Versand

Nach ein bis zwei Minuten können Sie danach den Mail-Client starten und das Mail-Konto auf neue E-Mails überprüfen. Eine neue E-Mail sollte jetzt in Ihrem Posteingang liegen.

3 Vorgehensweise bei der Softwareentwicklung

Im zweiten Kapitel dieses Buches wurden die grundlegende Syntax der Sprache PHP und eine Sammlung von Funktionen vorgestellt, die man im Alltag benötigt. Mit diesem „Wortschatz“ können Sie bereits prozedural und modular programmieren durch die Definition von eigenen Funktionen und deren Auslagerung in separaten Dateien, die über *require* oder *include* in das Skript eingebunden werden.

Dieses Kapitel hat keinen direkten Bezug zu der Sprache PHP. Es stellt stattdessen Methoden und Verfahren vor, wie man an die Softwareentwicklung herangeht. Dabei werden im ersten Teil strukturierte Vorgehensweisen zur Erstellung einer prozeduralen (PHP-)Anwendung vorgestellt. Es kommt häufig vor, dass ein meist junger Programmierer die Sprache PHP gerade erlernt hat und Probleme bei der Abwicklung seiner ersten Aufträge entstehen. Obwohl er die Sprache gut beherrscht und auch Prinzipien des Software-Engineerings kennt, werden oft Termine zur Fertigstellung der Software nicht eingehalten und/oder der Kunde hat sich die Anwendung bei der ersten Präsentation „ganz anders vorgestellt“. Der Kunde hat jedoch seine Wünsche nie konkret geäußert. Dieser Problematik widmet sich Kapitel 3.1.

Eine andere Dimension der Entwicklung ergibt sich dann, wenn ganze Programmierer-Teams an einem PHP-Projekt arbeiten und der Umfang des Projekts in seiner Gesamtheit gar nicht mehr von einer einzelnen Person überblickt werden kann. In diesen Fällen führt das „Hacken“ von Quellcode ohne eine weitere Organisation und Methodik unweigerlich zum Scheitern des Gesamtprojekts. Ebenso wird es problematisch, wenn die Software eine Größe erlangt, welche die Wiederverwendbarkeit einzelner Programmteile in anderen Projekten bedingt. Da PHP mittlerweile eine große Bekanntheit und auch einen guten Ruf als performante Skriptsprache ohne großen serverseitigen Aufwand erlangt hat, wird es immer häufiger für solche großen Projekte eingesetzt.

Man hat erkannt, dass die Prinzipien der Bildung von Funktionen und Unterprogrammen und der Aufspaltung von Funktionalität in separaten Dateien allein nicht ausreicht. Ebenso muss eine standardisierte Kommunikation mit den Kunden und den zukünftigen Anwendern der Software gefunden werden. Auch in dem gesamten Prozess der Software-Entwicklung sind seit der Idee der prozeduralen und modularen Programmierung viele neue Erkenntnisse und Methoden entstanden. Dazu zählen insbesondere agile Techniken, die gerade im Umfeld der Medien und Internetplattformen eine starke Verbreitung finden.

Auf dieser Basis ist das Konzept der objektorientierten Softwareentwicklung entstanden, das sich von der Analyse eines Geschäftsprozesses, der objektorientierten Modellierung der geschäftlichen Abläufe über den Entwurf eines technischen Modells bis hin zur objektorientierten Implementierung und Wartung der Anwendung erstreckt. Die dazu gehörenden Begrifflichkeiten und die Vorgehensweise werden in Kapitel 3.2 erläutert.

Besonders wichtig für einen Softwareentwickler sind die in Kapitel 3.2.2 eingeführten Definitionen. Denn zusätzlich zu den in Kapitel 2 dargestellten Funktionen von PHP muss die Sprache auch die Definitionen der Objektorientierung erfüllen. Wie PHP die in Kapitel 3.2.2 vorgestellten Konzepte umsetzt, wird dann im vierten Kapitel dieses Buches erläutert.

3.1 Prozedurale und modulare Programmierung

Im ersten Schritt wird auf die Vorgehensweise bei der „Programmierung im Kleinen" eingegangen, wie sie bis PHP4 üblich und auch erfolgreich war.

Der Kern der prozeduralen Programmierung besteht darin, eine Aufgabe in kleinere Teilprobleme aufzuteilen. Jeder Teil bildet eine Prozedur, die man mit PHP als *function* deklariert, die eine Menge von Variablen zur Eingabe benötigt, eine Verarbeitung vornimmt und eine Ergebnismenge als Ausgabe liefert. Dieses Vorgehen der *Eingabe-Verarbeitung-Ausgabe* wird als „EVA-Prinzip" bezeichnet. Die Teilung kann mehrfach erfolgen, da Funktionen wieder andere Funktionen aufrufen können. Auf der untersten Ebene werden nur PHP-eigene Funktionen und Anweisungen abgearbeitet.

Der Unterschied zwischen einer Prozedur und einem Modul besteht lediglich im Umfang der Funktionalität und damit in der Abstraktion zwischen der „Maschinendenkweise" und der Denkweise des Anwenders, der mit der PHP-Anwendung umgeht. Typische Aufgaben für Prozeduren sind beispielsweise

- Datenbankverbindung öffnen
- Datei schreiben
- Eingaben prüfen

Dies spielt sich auf einer technischen Ebene ab und ist daher von einem Programmierer leicht zu verstehen. Bei kleinen Problemen ist ein Programmierer in der Lage, eine Aufgabenstellung direkt auf diese Ebene herunterzubrechen.

Die Bezeichnungen für Softwaremodule abstrahieren vom PHP-Quellcode in Richtung des Anwenders. Wenn Sie einen Manager danach fragen, welche „Funktionen" denn das neue PHP-Portal besitzen soll, so kann er unter anderem antworten mit

- Artikel verwalten
- Kunden verwalten
- Bestellungen verwalten

Diese Funktionen sind so komplex, dass sie nicht direkt in einer einzigen PHP-Funktion abgearbeitet werden können. Die Artikel-, Kunden- und Bestellverwaltung stellen somit drei Module der neuen Anwendung dar. Wie diese Module funktionieren sollen, muss noch weiter hinterfragt werden. Als Programmierer einer PHP4-Anwendung würden Sie für jedes Modul ein Verzeichnis mit einer Sammlung von PHP-Dateien anlegen. Die Artikelverwaltung würde unter anderem über Dienste verfügen wie

- einen neuen Artikel anlegen
- nach Artikel suchen
- einen Artikel anzeigen
- einen vorhandenen Artikel ändern
- einen Artikel aus dem Angebot entfernen

Für jedes dieser Teilmodule können Sie Funktionen verwenden, die Sie auf prozeduraler Ebene entwickelt haben. So erfordert das Anlegen eines neuen Artikels die Darstellung einer Eingabemaske für einen neuen Artikel, die anschließende Prüfung der Eingaben des Benutzers, das Öffnen einer Verbindung zur Datenbank, das Übertragen der eingegebenen Daten zur Datenbank und abschließend die Darstellung einer HTML-Seite für den Benutzer, ob das Anlegen nun erfolgreich war oder nicht. Damit haben Sie eine Funktionalität der Artikelverwaltung realisiert.

Das Herunterbrechen von Anforderungen an eine Software auf Quellcodeebene wird als *Top-Down*-Vorgehensweise bezeichnet, die heutzutage am weitesten verbreitet ist. Eine andere Bezeichnung dafür ist das Grundprinzip des *Divide-And-Conquer* (teile und herrsche) als Methode der Algorithmik in der Informatik. Es beschreibt das Prinzip, eine Aufgabe bzw. eine geforderte Funktionalität in kleinere Einheiten zu zerteilen und sie nacheinander mit den Mitteln der Programmiersprache PHP abzuarbeiten.

Wie viel Zeit werden Sie für die Realisierung einer Artikelverwaltung benötigen? Multipliziert mit Ihrem Stundensatz: Wie viel Geld wollen Sie von Ihrem Kunden dafür verlangen? Für eine Aufwandschätzung brechen Sie in der Regel zunächst die Anforderungen Ihres Kunden im Vorfeld in Prozeduren herunter und beginnen dann, die geforderten Module aus den elementaren Prozeduren zusammenzusetzen. Falls sie bislang nur wenige oder keine Softwareprojekte durchgeführt haben und eine Aufwandschätzung vom potentiellen neuen Kunden gefordert wird, so neigen viele Entwickler dazu, den Aufwand bereits für kleine Projekte drastisch zu unterschätzen!

Profitipp

Schätzen Sie als Anfänger den Aufwand (Zeit und Kosten) für ein zukünftiges Softwareprojekt ein und multiplizieren Sie Ihre Einschätzung mindestens mit dem Faktor 3. Dann bestehen gute Chancen, dass Sie bei dem Projekt zumindest keinen Verlust machen.

Eine Vorgehensweise, die bei einer Sammlung von Prozeduren beginnt und bei der Funktionalität für den Anwender endet, wird als *Bottom-Up*-Strategie bezeichnet. Diese Vorgehensweise wird häufig dann eingesetzt, wenn bereits eine Vielzahl von Prozeduren bei Projektbeginn fertig vorliegt, weil sie aus alten Projekten wiederverwendet werden kann.

Profitipp

Die Wiederverwendbarkeit von Quellcode spielt in der Softwareentwicklung eine immer größere Rolle, um Kosten und Zeit für die Erstellung einer individuellen Anwendung einzusparen. Daher wird die Wiederverwendbarkeit in der Objektorientierung (Kap. 3.4 und Kap. 4) stärker berücksichtigt. Programmieren Sie also in jedem Fall so, dass möglichst viel Ihres Quellcodes problemlos durch Kopieren in zukünftigen Projekten anwendbar ist.

Gerade bei den ersten eigenständigen Projekten haben viele Entwickler Schwierigkeiten. Wie man von den Vorstellungen Ihres (potenziellen) Kunden zu einer Softwarelösung gelangt, ist in der Informatik eine eigene Wissenschaft geworden, die als „Software-Engineering" bezeichnet wird, was man im Deutschen meist mit Softwaretechnik übersetzt. Diese Wissenschaft hat über die Jahre eine Reihe von Modellen und Vorgehensweise hervorgebracht, die jeder (PHP-)Entwickler kennen sollte.

In der kontinuierlichen Weiterentwicklung der Modelle wurde insbesondere die zunehmende Anzahl an Projektbeteiligten aus verschiedenen Fachbereichen, die zunehmende Projektgröße und -komplexität berücksichtigt. Während im weiteren Verlauf dieses Kapitels traditionellere Vorgehensweisen vorgestellt werden, die bei bestimmten Projekten immer noch ihre Berechtigung haben, widmet sich Kapitel 3.2 der Objektorientierung mit den aktuellen agilen Methoden der Softwareentwicklung, die den Kern dieses Buches darstellen.

3.1.1 Typische Projektstruktur

Die in Kapitel 3.1.2 vorgestellten Modelle eignen sich insbesondere für kleine Projekte, bei denen Sie (nahezu) der alleinige Entwickler sind. In der Softwaretechnik werden Projekte als „klein" eingestuft, wenn sie in ca. 2 Personenjahren erledigt werden können. Während das Wasserfallmodell ausschließlich für diese Art von Projekten einzusetzen ist, eignen sich die im Folgenden vorgestellten Vorgehensweisen nach dem Spiral- und dem V-Modell auch bereits für mittelgroße Projekte.

Typische kleine PHP-Projekte erzeugen bis zu 10 000 Zeilen Quellcode und bestehen normalerweise aus folgenden Aufgaben:

- Erstellung einer Präsentation für ein kleines oder mittelständisches Unternehmen, bei dem der Inhalt von einzelnen Seiten aus einer Datenbank gespeist wird.
- Erstellung eines kleinen B2B-, B2C- oder B2E-Portals (Business-to-Business, Business-to-Consumer oder Business-to-Employee) mit Login.

Eine typische B2B-Anwendung ist beispielsweise ein Datentransfer von einer Artikel-Datenbank in eine andere Artikeldatenbank mittels PHP, wobei beide Datenbanken eine verschiedene Struktur aufweisen.

Eine typische B2C-Anwendung ist es, wenn ein kleiner Laden seine Artikel auch im Internet anbieten möchte. Bestellungen sollen informell möglich sein, indem ein Kunde sich registriert und seine E-Mail-Adresse angibt. Zur weiteren Bearbeitung der Bestellung wird dann eine E-Mail an den Ladenbesitzer versendet.

Als B2E-Anwendung ist unter anderem eine „Arbeitszeiterfassung per Internet" denkbar. Eine Firma, die Mitarbeiter verleiht, möchte die Arbeitszeiterfassung online ermöglichen, indem nach einem Login des Mitarbeiters in vorgefertigten dynamischen Formularen die Arbeitszeiten eingegeben werden; der Mitarbeiter kann seine Zeiten des letzten Jahres einsehen.

Hinweis

Eine gute Übung zum Einstieg in PHP besteht darin, dass Sie sich eine der oben genannten Aufgaben aussuchen und versuchen, die geforderte Funktionalität zu realisieren. Die PHP-Kenntnisse aus Kapitel 2 sollten dazu ausreichen. Schätzen Sie vorher ab, wie lange Sie für die Realisierung einer Funktion (Beispiel: „Einen neuen Artikel in der MySQL-Datenbank anlegen") benötigen und messen Sie, wie lange Sie tatsächlich dafür gebraucht haben. Dies gibt ein erstes Gefühl für die notwendige Aufwandschätzung.

Als Referenzprojekte dieser Größenordnung kann ich meine eigene Diplomarbeit und Master Thesis empfehlen. Im Rahmen der Diplomarbeit mit dem Thema „Konzeption und Realisierung eines Online-Praktikums sowie der Basis eines Internet-Portals für das TDI-Labor" (*http://www.frankdopatka.de/studium/gm/2002_diplom.pdf*, 15MB) wurde bereits im Jahre 2002 die Basis für ein Internetportal für Studenten geschaffen, die ein Onlinepraktikum an einer Fachhochschule absolvieren sollen. Neben der Benutzerverwaltung wird die Anbindung eines Mikro-Controllers über das PHP-Portal beschrieben.

Die Master-Thesis mit dem Titel „Konzeption und Realisierung eines Mitarbeiterportals mit Arbeitszeiterfassung und dezentralem Datenbankabgleich" aus dem Jahre 2003 (*http://www.frankdopatka.de/studium/koeln/2003_master.pdf*, 9 MB) zeigt die Wiederverwendung des Logins und die Erweiterung auf verschiedene Benutzergruppen. Hierbei handelt es sich um ein klassisches B2C-Portal.

Weiterhin typisch für kleine Projekte ist es, dass Sie zur Realisierung lediglich einfache Werkzeuge benötigen. Dies sind ein Texteditor für die Erstellung der PHP-Dateien sowie ein XAMPP-Server, der auf jedem gängigen PC oder Laptop lauffähig ist.

3.1.2 Ablauf eines Projekts

Während sich der theoretische Teil der Informatik wie die formalen Sprachen aus der Mathematik ableiten, stammt ein anderer Teil der Informatik aus den Ingenieurwissenschaften. Der Fokus eines Ingenieurs liegt im Gegensatz zu einem Theoretiker auf der Anwendung von Wissen in Projekten.

Die Sprache PHP (Kap. 2) kann man sich privat aneignen und damit kleine Problemstellungen lösen. Wenn Sie als Freiberufler für einzelne andere Personen eine kleine PHP-Anwendung schreiben, gehen Sie vermutlich nur mit gesundem Menschenverstand und intuitiv vor. Aus diesem Grunde wurde die Erstellung eines Programms früher als „Ingenieurskunst" bezeichnet. Sie sind in der Lage, aus einer Problemstellung eines Kunden heraus nach einigen Gesprächen PHP-Dateien mit Verzweigungen, Schleifen,

HTML-Formularen und Datenbankanbindungen zu erstellen, sodass die Wünsche des Kunden erfüllt werden. Dies zeichnet einen Entwickler bzw. einen Programmierer aus.

Seit den letzten 30 Jahren wurden die Problemstellungen jedoch zunehmend komplexer. Man sehnte sich nach einer ultimativen Vorgehensweise, einer Anleitung bzw. nach einem Algorithmus, wie man jedes Problem eines jeden Kunden lösen kann. Die in diesem Kapitel beschriebenen Methoden repräsentieren den Kenntnisstand zwischen 1970 und 2000.

Nun denken Sie vielleicht, dass diese Methoden doch völlig veraltet sind und daher keinerlei praktische Bedeutung mehr haben. Dies wird in der Wissenschaft auch häufig so dargestellt. Die Kenntnis dieser Methoden unterscheidet jedoch einen Entwickler von einem Hobbyprogrammierer, der sich nur aus Interesse mit der Sprache PHP beschäftigt. Außerdem bieten diese Vorgehensweisen für bestimmte Projekte, u. a. kleinere Projekte mit wenigen Beteiligten, eine sinnvolle Anleitung, die man zumindest im Hinterkopf halten sollte. Bereits im Wasserfallmodell von 1970 wurden „Phasen" erwähnt, die eine Softwareentwicklung durchläuft. Diese Phasen haben heutzutage noch immer ihre Bedeutung. Viele modernere Vorgehensweisen haben die Komplexität der Anwendungsentwicklung besser verstanden als ältere Modelle der Softwareentwicklung und versuchen, die Realität genauer abzubilden. Sie können also davon ausgehen, dass die in diesem Kapitel vorgestellten Modelle in Bezug zur Realität vereinfacht und auch idealisiert sind. Sie eignen sich jedoch hervorragend als Einstieg in das Projektmanagement der Anwendungsentwicklung mit PHP.

Das Wasserfallmodell

Das Wasserfallmodell ist das erste bekannte lineare Modell, das Ihnen eine Anleitung für die Realisierung eines (PHP-)Projekts liefert. Es beschreibt eine Vorgehensweise, wie Sie von der Idee einer Anwendung bis hin zur Abgabe und der Betreuung des fertigen Produkts gelangen.

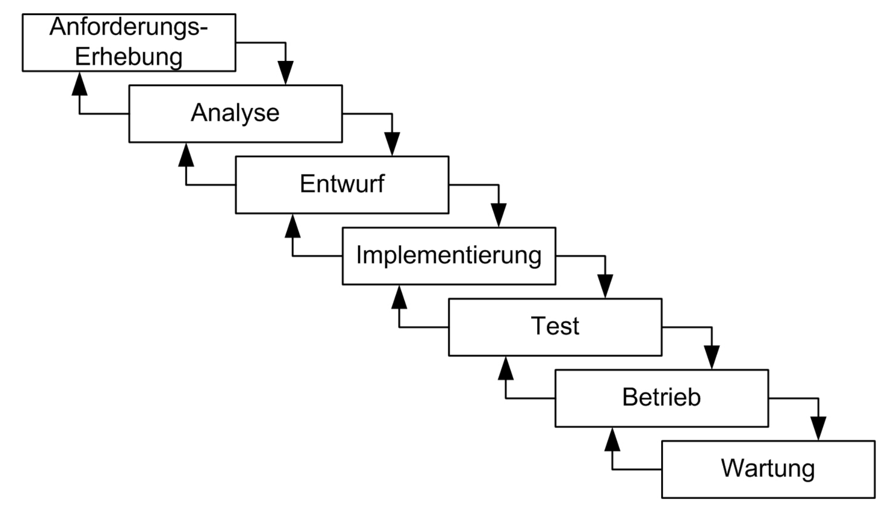


Abbildung 3.1: Das Wasserfallmodell aus dem Jahre 1970

Der erste Schritt besteht in der Anforderungserhebung, also dem Wunsch nach einer neuen Software. Diese Idee kommt in der Regel vom Kunden oder wird von Ihnen durch geschicktes Marketing erzeugt, indem Sie den Bedarf beim Kunden wecken. Wenn Sie möglichst früh an der Anforderungserhebung teilhaben, können Sie den Verlauf des Projektes aus Sicht des Entwicklers bereits in die von Ihnen gewünschte Richtung lenken. Ziel der Anforderungserhebung ist es, die Anforderungen des Auftraggebers an das zu entwickelnde System zu ermitteln.

In der Analyse der Anforderungen formalisieren Sie die Wünsche Ihres Kunden. Sie halten fest, welche Funktionalität er gerne in Ihrer PHP-Anwendung sehen würde und in welchen Schritten er Ihre Anwendung nacheinander bedient. Bei einem Onlineshop wäre beispielsweise folgende Reihenfolge denkbar:

1. Der Kunde sucht in den Artikeln nach verschiedenen Kriterien
2. Der Kunde möchte gern den ersten Artikel kaufen
3. Der Kunde registriert sich mit Namen, Vornamen, Anschrift usw. oder loggt sich ein
4. Der Kunde legt eine Anzahl des gewünschten Artikels in seinen Warenkorb
5. Wiederholung der Schritte 1, 2 und 5; der Kunde bleibt eingeloggt
6. Der Kunde wechselt zum Warenkorb
7. Der Kunde bestätigt die Bestellung
8. Der Kunde loggt sich aus

Die Anforderungsanalyse beschreibt also die geschäftlichen Abläufe, die mithilfe der Software realisiert werden sollen. Sie endet mit dem Lastenheft. Das Dokument wird von Ihrem Kunden unterschrieben. Es beschreibt die vollständigen Forderungen Ihres Auftraggebers an Ihre Lieferungen und Leistungen. Das Lastenheft gehört Ihrem Kunden. In der Regel erhalten Sie eine Kopie mit der Anfrage, wie teuer denn die Realisierung wäre. Dies müssen Sie so schätzen, dass Sie mit Sicherheit keinen Verlust machen und dennoch den Auftrag erhalten, was äußerst schwierig ist und viel Projekterfahrung erfordert. Ihr Kunde kann nämlich das Lastenheft auch in einer Ausschreibung verwenden und an andere mögliche Programmierer versenden.

In der Entwurfsphase überlegen Sie, wie Sie gedenken, die Anforderungen des Kunden umzusetzen. Sie modellieren die Lösung hier theoretisch und erwähnen auch, dass Sie PHP in einer WAMP-Architektur verwenden möchten. Dazu erklären Sie, wie Sie die wichtigsten beschriebenen geschäftlichen Abläufe abbilden wollen. Diese Beschreibungen enden in einem Pflichtenheft. Es umfasst nach DIN 69905 die „vom Auftragnehmer erarbeiteten Realisierungsvorgaben aufgrund der Umsetzung des vom Auftraggeber vorgegebenen Lastenhefts". Generell endet jede Phase in einem Meilenstein, der mit einem Ergebnis (sei es ein Dokument oder eine Anwendung) verbunden ist.

Sie legen das Pflichtenheft idealerweise zusammen mit Ihrer Kostenkalkulation und einem möglichen Abgabetermin Ihrem Kunden vor und Kunde gibt Ihnen daraufhin den Auftrag.

In der Phase der Implementierung ziehen Sie sich vom Kunden zurück in Ihr Büro und programmieren die Lösung. Das schaffen Sie idealerweise in der vorgeschriebenen Zeit, denn Sie haben sich ja in der Entwurfsphase bereits die notwendigen Gedanken gemacht, wie Sie zur Lösung kommen.

Wenn Sie mit der Implementierung fertig sind, testen Sie Ihre Software zunächst selbst und dann zusammen mit dem Kunden. Dieser macht abschließend eine Abnahme und Ihre Anwendung geht in Betrieb. Die Abnahme kann beispielsweise mit einem Projektabschlussbericht nach DIN 69901 enden, der die „zusammenfassende, abschließende Darstellung von Aufgaben und erzielten Ergebnissen, von Zeit-, Kosten- und Personalaufwand sowie gegebenenfalls von Hinweisen auf mögliche Anschlussprojekte" enthält.

Ab und zu kommt es vor, dass einige kleine Fehler erst später gesehen werden. Diese beheben Sie dann. Oder der Kunde möchte einige Erweiterungen an Ihrem PHP-Shop, wie eine zusätzliche Artikelansicht oder ein neues Design nach zwei Jahren. Solche Änderungsanforderungen (Change Requests) rechnen Sie separat ab.

Wenn Sie bereits etwas Erfahrung mit Aufträgen in der freien Wirtschaft haben, werden Sie jetzt vielleicht denken: „Schön, wenn es immer so wäre." Oft stellt man erst später fest, dass die Annahmen in der vorherigen Phase, die ggf. ja sogar unterschrieben wurde, so nicht oder nicht ganz zutreffen. Das Wasserfallmodell lässt es zu, dass man jeweils eine Phase, also einen Schritt, zurückgehen, die Änderungen einpflegen und neu kalkulieren kann. Bei schwierigen Kunden kann natürlich die Aussage kommen: „Wie, mehr Geld? Nein, das haben Sie so unterschrieben! Unsere Anforderungen waren von Anfang an klar! Sie haben es falsch verstanden oder nicht gefragt. Sie sind doch der Experte!" Dies endet oft mit einem unzufriedenen Kunden und einem Rechtsstreit.

Das Wasserfallmodell wird in der Softwaretechnik oft kritisiert, da der Kunde erst sehr spät eine Anwendung sieht und seine Wünsche nur zu Beginn des Projekts einfließen lassen kann. Es hat sich seit seiner Veröffentlichung von Winston Royce im Jahre 1970 herausgestellt, dass viele Kunden zu Beginn des Projekts ihre eigenen Anforderungen gar nicht genau kennen und deshalb auch nicht für einen Programmierer spezifizieren können. Erst wenn sie eine Anwendung (einen Prototyp) sehen, machen sich die Entscheider Gedanken darüber. Und daraus entwickeln sich wieder neue Ideen für Funktionen.

Das Wasserfallmodell ist durch das Lasten- und Pflichtenheft sehr bürokratisch und formal. Dies hat für den Kunden den Vorteil, dass er sehr früh einen Preis genannt bekommt. So kann er besser mit seinem Budget umgehen und Ausschreibungen vergleichen. Für Sie als (freiberuflicher) Entwickler ist dies jedoch risikoreich. Man sagt, dass bei einer Wasserfall-Vorgehensweise die Phasen Implementierung bis incl. Betrieb 80 % des Gesamtaufwands ausmachen, die Analyse bis zum Design die restlichen 20 %.

Im Allgemeinen ist das Wasserfallmodell nur bei sehr kleinen Projekten durchführbar und auch nur dann, wenn Sie den Kunden und dessen Wünsche bereits im Vorfeld gut abschätzen können. Als Programmiertechnik ist die prozedurale Vorgehensweise ausreichend, die im zweiten Kapitel vorgestellt worden ist.

Das Spiralmodell

Dass man die Anforderungen nicht vollständig im Voraus kennt und der Kunde bei der Entwicklung der (PHP-)Anwendung öfter eingebunden werden muss, indem Sie ihm Prototypen der Software präsentieren, wurde im Spiralmodell berücksichtigt. Dieses Modell wurde 1988 von Barry Boehm veröffentlicht, also ganze 18 Jahre nach dem Wasserfallmodell. Der Kern dieses Modells ist iterativ; die bislang beschriebenen Phasen werden demnach ständig wiederholt.

Nachdem der Kunde den Bedarf für die Software erkannt hat, startet im Spiralmodell das Projekt. Sie erstellen möglichst schnell einen kleinen Prototyp oder präsentieren dem (potenziellen) Kunden ein bereits durchgeführtes ähnliches Projekt als Diskussionsgrundlage. Nun macht man sich Gedanken über die betrieblichen Abläufe (vgl. Analyse im Wasserfallmodell). Das Ziel ist die Erstellung des nächsten, etwas passenderen Prototyps der Anwendung. So ergeben sich mit der Zeit die Anforderungen an die zu realisierende Anwendung bzw. an das Produkt.

Die Achsen des Spiralmodells gliedert die im Wasserfallmodell beschriebenen Phasen in vier, sich wiederholende Aktivitäten:

1. Bestimmung von Zielen, Identifikation von Alternativen und Beschreibung von Rahmenbedingungen und Beschränkungen
2. Bewertung der Alternativen und das Erkennen, Abschätzen und Reduzieren von Risiken, z. B. durch Simulationen oder Prototyping
3. Realisierung und Überprüfung der nächsten Produktstufe
4. Planung des nächsten Zyklus zur Fortsetzung des Projekts

Insbesondere in der dritten Aktivität finden Sie sich in der Rolle des Entwicklers wieder. In der letzten Spirale ist dort der Feinentwurf, die Kodierung der neuen Komponenten mit anschließendem Test und der Integration in die bislang erstellte Anwendung sowie die abschließende Einführung der Software dargestellt. Wichtig ist, nochmals zu erwähnen, dass dies nicht einmalig für das gesamte Projekt, sondern iterativ für jede Realisierung einer Produktstufe gilt. So wird mit der Zeit aus dem Prototyp eine fertige Anwendung, die Sie in Kooperation mit dem Kunden mit dem Management erstellen.

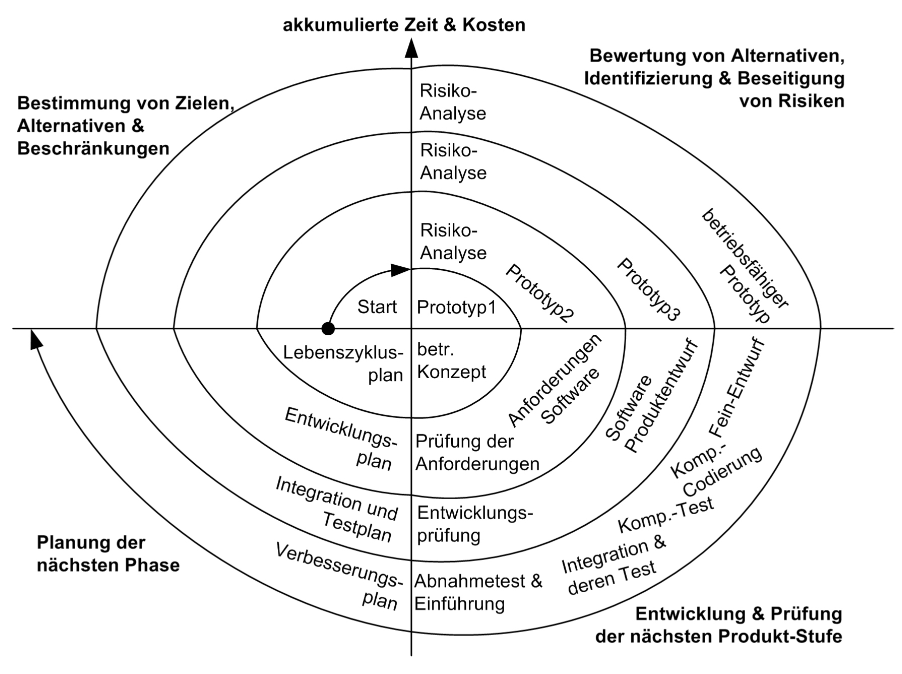


Abbildung 3.2: Das Spiralmodell aus dem Jahre 1988

Als Nachteil des Spiralmodells gilt, dass es nicht endet. Mit jedem Durchlauf erhöhen sich für Ihren Kunden die Kosten und es vergeht natürlich auch eine Menge Zeit. Der Kunde erfährt nicht im Vorfeld, wie teuer denn die Realisierung seiner Wünsche ist. Sie als (freiberuflicher) Entwickler können also im Vergleich zum Wasserfallmodell nicht zu einem frühen Zeitpunkt ein verbindliches Angebot abgeben. Dies ist aber auch andererseits fast nie möglich. Lässt sich der Kunde auf dieses Vorgehen ein, so wird er das Projekt dann beenden, wenn ihm die Ressourcen Zeit und Budget ausgehen. Im Idealfall geben Sie dem Kunden eine Planung für den nächsten Zyklus an und lassen ihn selbst entscheiden.

Das V-Modell

Nahezu zeitgleich mit dem Spiralmodell wurde 1986 in Deutschland das V-Modell, zunächst insbesondere im militärischen Umfeld, entwickelt. Neben der Kostenoptimierung steht beim V-Modell die Softwarequalität im Vordergrund mit der Forderung, eine Software vor ihrer Inbetriebnahme ordentlich zu testen. Das V-Modell hat noch heutzutage eine besondere Bedeutung, da viele öffentliche Ausschreibungen ein Vorgehen nach diesem Modell verlangen. In einem solchen Fall sollten Sie beispielsweise bei der Bewerbung auf ein PHP-Portal für eine staatliche Institution Ihre Vorgehensweise entsprechend beschreiben.

Im Gegensatz zum Wasserfallmodell werden im V-Modell nur Aktivitäten und Ergebnisse definiert und keine strikte zeitliche Abfolge gefordert. Insbesondere fehlen die Abnahmen, die ein Phasenende definieren.

Die linke Hälfte der V-Darstellung erinnert an das Wasserfallmodell, wobei eine Kleinigkeit leicht übersehen wird: Die X-Achse des V-Modells stellt den zeitlichen Ablauf dar. Während also die Anforderungen an die Benutzer noch spezifiziert werden, werden bereits die Anforderungen an die Entwickler beschrieben. Auch die Problemstellung muss noch nicht vollständig ausgearbeitet sein. Und es können bereits erste kleine Teile des Systems entworfen werden. Die Phasen gehen also fließend ineinander über.

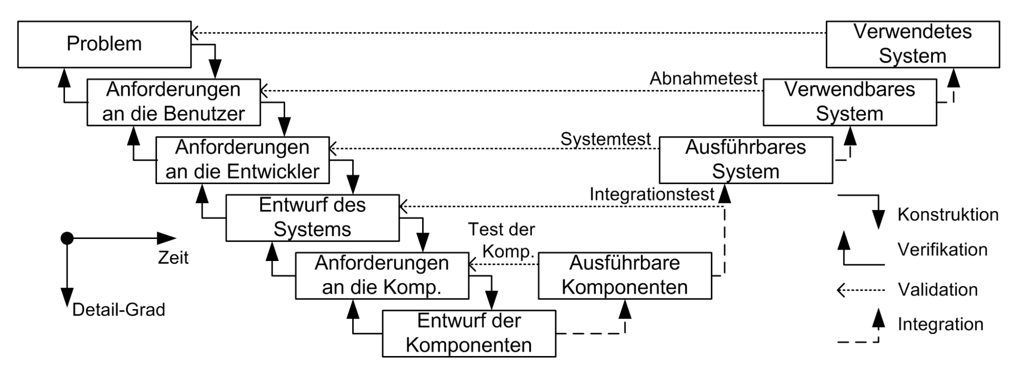


Abbildung 3.3: Das deutsche V-Modell aus dem Jahre 1986

Die Y-Achse des V-Modells beschreibt, wie tief man in die technische Realisierung blickt. Während der Bedarf und die Anforderungen meist vom Management und den Benutzern in einer sehr groben und funktionsorientierten Weise beschrieben werden („Was brauchen wir?“, „Was soll die Anwendung können?“), betrachtet der Entwickler die ein-

zelnen Komponenten der zu entwickelnden Software („Öffnen einer Datenbankverbindung...“, „HTML-Formular ausfüllen und prüfen...“). Während einzelne Module noch vom Entwickler selbst getestet werden, wird die gesamte Anwendung wiederum vom Management und den späteren Benutzern getestet.

Der rechte Teil des V-Modells beschreibt also die Tests der Software, sowohl auf der Ebene der einzelnen Komponenten als auch auf Systemebene. Die Tests der einzelnen Komponenten werden auch als „Unit-Tests“ bezeichnet und sind heutzutage ebenso automatisierbar wie die Integrationstests, bei denen die Komponenten zu dem Gesamtsystem zusammengebaut werden. Im Umfeld der Integrationstests ist das Schlagwort „Continuous Integration“ zu nennen. Der Trend geht dahin, in Verbindung mit einer Versionsverwaltung in festen Zeitabständen den aktuellen Stand der Anwendung zu erzeugen. Dieser kann dann bereits als Prototyp von zukünftigen Benutzern eingesehen und kommentiert werden. Für die Unit-Tests empfiehlt sich das Tool PHPUnit, für das Management der kontinuierlichen Integration eines (größeren) PHP-Projekts das Servertool Xinc. Stets testet man gegen die entsprechende Spezifikation; den Quellcode gegen die Anforderungen an den Quellcode, das ausführbare System gegen die Anforderungsbeschreibung an die Entwickler sowie bei der Abnahme des fertigen Produkts gegen die Funktionen, die von den Benutzern und dem Management gefordert wurden.

Auffällig ist bereits, dass die Tests als „Validation“ und die Rückschritte in der Spezifikation im linken Teil des Modells als „Verifikation“ bezeichnet werden. Wo liegen die Unterschiede zwischen den beiden Begriffen? Bei der Validierung handelt es sich um eine Prüfung, ob ein Entwicklungsergebnis die individuellen Anforderungen bezüglich einer speziellen beabsichtigten Nutzung erfüllt. Es wird also geprüft, ob das realisiert wurde, was vom Kunden gefordert war. Die Verifikation ist ebenfalls eine Prüfung. Hier wird geprüft, ob die Ergebnisse einer Entwicklungsphase den Vorgaben der Dokumentation, die den Input dieser Phase bildeten, entsprechen.

Neben den erstellten Softwarekomponenten werden im V-Modell weitere Produkte definiert, bei denen es sich um Dokumente zur Spezifikation handelt. Jedes Produkt durchläuft die vier Zustände *geplant*, *in Bearbeitung*, *vorgelegt* und *akzeptiert*. Wenn Ihr Kunde ein vorgelegtes Produkt (sei es als Schriftstück oder als Funktionalität im Prototyp der Software) nicht akzeptiert, wird es nachgebessert.

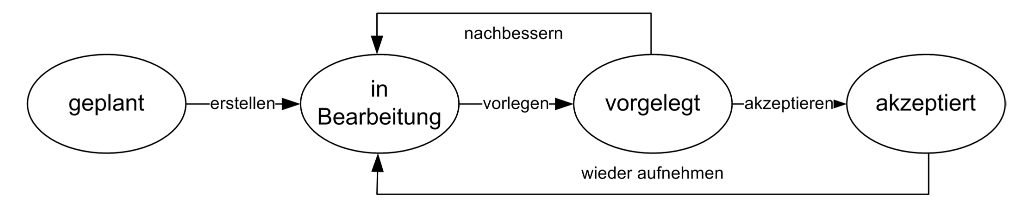


Abbildung 3.4: Zustände von Produkten im V-Modell

Viel bedeutender ist jedoch die Aussage in Abbildung 3.4, dass bereits akzeptierte Produkte wieder aufgenommen werden können. Wie ist es möglich, dass etwas Fertiges nochmals bearbeitet wird? Bei der Erstellung des V-Modells wurde erstmals beachtet, dass die Anforderungen an eine Software selbst dem Kunden nicht von Beginn an bekannt sind. Bei einem länger andauernden Projekt ist es ebenso normal, dass sich

Anforderungen mit der Zeit ändern. Dies gibt dem Modell eine weitaus größere Dynamik als dem formalen und idealisierten Wasserfallmodell.

Gleichzeitig werden die benötigten Ressourcen wesentlich schwieriger kontrollierbar. Wie können das Ende des Projekts und die Kosten im Vorfeld bestimmt werden, wenn die Anforderungen noch gar nicht bekannt sind und sich sogar ändern können? Um diesen Problemen zu entgegnen, beinhaltet der Kern des V-Modells vier Vorgehensbausteine, die heutzutage als Managementaktivitäten gesehen werden. Dabei handelt es sich um

1. das Projektmanagement, das den Überblick über Kosten und Termine behält,
2. die Qualitätssicherung, um die Stufen der Validation und Verifikation zu garantieren,
3. das Konfigurationsmanagement, das Regeln auf den Produktlebenslauf anwendet, und
4. das Problem- und Änderungsmanagement zur Umsetzung neuer Strukturen, Prozesse oder Verhaltensweisen in der zu erstellenden Anwendung. Hier wird insbesondere auch unterschieden, ob es sich bei Unzulänglichkeiten der bisher erstellten Anwendung um Fehler handelt (die in der Regel innerhalb des bestehenden Budgets behoben werden müssen) oder um neue Anforderungen (für die normalerweise neue Budgets freigegeben werden müssen).

Am V-Modell wird meist kritisiert, dass die Testphasen erst spät beginnen. Werden Unzulänglichkeiten erst beim Abnahmetest bekannt, so erfordert dies zumeist eine umfassende Änderung der Anforderungen bis hin zu der entwickelten Anwendung.

Abbildung 3.5 zeigt die Abhängigkeit der entstehenden Zusatzkosten von der Phase des Softwareprojekts: Je später ein Fehler erkannt wird, desto höher werden die Kosten, um ihn zu beheben. In laufenden Anwendungen sind sogar rechtliche Schritte Ihres Kunden gegen Sie möglich, falls beispielsweise ein Produktionsausfall entsteht, weil Ihr Kunde zwei Tage lang keine Bestellungen über sein B2B-Portal entgegennehmen kann.

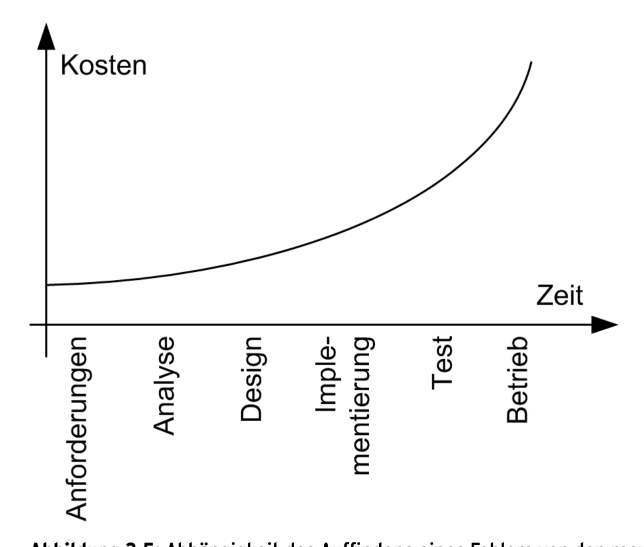


Abbildung 3.5: Abhängigkeit des Auffindens eines Fehlers von den resultierenden Kosten zu seiner Behebung

3.1.3 Erstellung der 3-Schichten-Architektur

Eine weit verbreitete Strukturierung einer zu erstellenden Anwendung besteht in der Aufteilung in drei Schichten. Die Datenzugriffsschicht kapselt den Zugriff auf die Datenbank, der die Verwaltung der Verbindung zum Datenbankserver, die SQL-Abfragen und die Auswertung der Resultsets beinhaltet. Die PHP-Dateien, die den Datenzugriff realisieren, bieten elementare Dienste an, wie das Auslesen von Kundendaten, das Anlegen eines Neukunden, die Suche nach einem Kunden oder die Rückgabe aller Bestellungen eines Kunden K. Solche Dienste sind, sofern sie unabhängig voneinander programmiert wurden, wiederverwendbar. Dadurch können bei zukünftigen Projekten Kosten und Zeit gespart werden.

Die Fachlogik bildet die mittlere Schicht der Anwendung. Hier werden die Dienste bzw. die Funktionen realisiert, die Ihr Kunde verlangt hat. Die Fachlogik greift auf die Funktionen der Datenzugriffsschicht zu, wodurch Anfragen auf die Datenbank ausgelöst werden. Die Ergebnisse der Abfragen werden dann in der Fachlogik interpretiert und weiter verarbeitet. Gleichzeitig werden die Eingaben von der Präsentationsschicht – die das Graphical User Interface (GUI) realisiert – entgegennimmt und auf Plausibilität prüft. Typische Dienste der Fachlogik sind Verwaltungsmodule wie eine Kunden-, Artikel- oder Rechnungsverwaltung mit den entsprechenden Funktionalitäten.

Oft wirken die Aufgaben der Präsentationsschicht im Vergleich zu den anderen Schichten sehr einfach und schon trivial. Es ist jedoch keine leichte Aufgabe, ein gutes Benutzerinterface zu schaffen. Einerseits muss das Design gerade im Internetumfeld an die Gepflogenheiten Ihres Kunden angepasst sein, um ein einheitliches Erscheinungsbild von Internetauftritt, Visitenkarten und Briefköpfen zu ereichen (Stichwort: Corporate Identity). Zusätzlich müssen sich die Eingaben von Daten an den Geschäftsprozess des Kunden, also an die alltägliche Arbeit, anpassen, da die Benutzer die Anwendung sonst unzufrieden sind, was wiederum für mangelnde Akzeptanz sorgt. Weitere Merkmale der Präsentations-Schicht liegen in einer ergonomischen Bedienung. So sollten alle Steuerelemente wie Eingabe-Felder und Schaltflächen über die gesamte Anwendung einheitlich positioniert sein; bei mehr als 100 Masken kann dies problematisch sein. Abbildung 3.6 stellt die Aufteilung der drei Schichten nochmals grafisch dar.

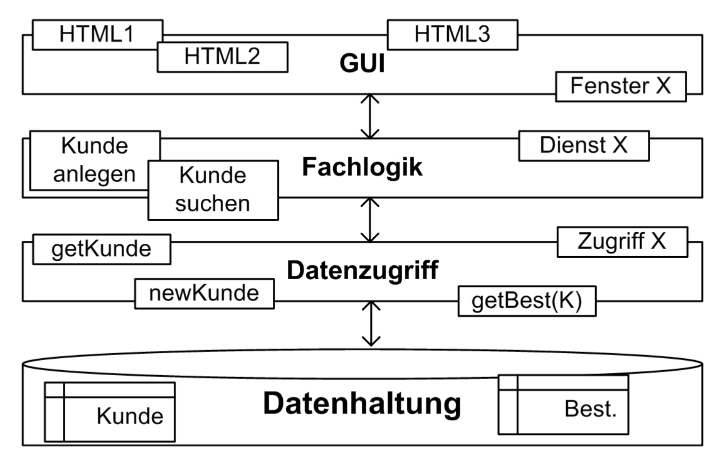


Abbildung 3.6: 3-Schichten-Architektur der Anwendungsentwicklung

Für Sie als PHP-Entwickler stellt sich die Frage, wie Sie die drei Schichten realisieren können, da Sie ja „nur" PHP-Dateien schreiben. Die Antwort liegt in der sinnvollen Strukturierung Ihrer Anwendung.

Datenhaltung und Datenzugriff

So besitzen alle Dateien, die zum Datenzugriff gehören, keine aktive Funktionalität zum Benutzer hin. Sie werden ausschließlich von der Fachlogik aufgerufen. Daher können Sie in *dz_xxx.inc.php* umbenannt werden, z. B. in *dz_kunde.inc.php* oder in *dz_bestellung.inc.php*. Diese Dateien werden also ausschließlich von anderen PHP-Dateien inkludiert und kapseln die Zugriffe auf die Datenbank inklusive alle SQL-Anweisungen. Dies hat den weiteren Vorteil, dass die Sprache SQL auf eine kleine Menge von Dateien beschränkt bleibt, was für Übersicht sorgt. Außerdem würde es einen Webdesigner verwirren, wenn er bei der Anpassung des Designs auf SQL-Statements stößt.

Falls Sie auf keine bestehende Datenbank zugreifen, gehört das Erstellen der Datenbanktabellen mit zu Ihren Aufgaben. Als Hilfsmittel dient die ER-Modellierung mit dem Ziel, Namen von Datenbanktabellen, deren Inhalte und Beziehungen zu anderen Tabellen zu ermitteln. Die ER-Modellierung hat eine enge Verwandtschaft mit der Modellierung von Klassen in der Objektorientierung, Kapitel 3.2. Jeder Datensatz einer Tabelle sollte über einen eindeutigen Identifier angesprochen werden, den Primärschlüssel. Dies sorgt für einen robusten, schnellen Zugriff. Typische Primärschlüssel sind die Kundennummer der Kundentabelle, die Rechnungsnummer der Rechnungstabelle, die ISBN-Nummer der Büchertabelle und die Fahrgestellnummer der Fahrzeugtabelle. Datenbanktabellen können in drei Arten von Beziehungen zueinander stehen:

- *1:1-Beziehung*:Ein Fahrzeugführer kann genau einen Führerschein besitzen und ein Führerschein ist genau einem Fahrzeugführer zugeordnet. In einem solchen Fall wird die Personalnummer mit in den Datensatz des Führerscheins gespeichert. Wird ein Primärschlüssel eines Datensatzes in einer anderen Tabelle gespeichert, wird dies als Fremdschlüssel bezeichnet (Kap. 2.2). In der Praxis führt man 1:1-Beziehungen meist zu einer einzigen Tabelle zusammen.
- *1:n-Beziehung*: Einem Kunden können beliebig viele Rechnungen zugeordnet werden, eine Rechnung gehört aber genau zu einem Kunden. Auch hier wird die Kundennummer mit in den Datensatz der Rechnung gespeichert; man arbeitet also wieder mit einem Fremdschlüssel.
- *n:m-Beziehung*: Ein Lehrer kann verschiedene Fächer unterrichten und ein Fach kann von verschiedenen Lehrern unterrichtet werden. Eine solche Beziehung wird über eine Hilfstabelle realisiert, die nur die beiden Primärschlüssel (hier: Lehrer-ID und Fach-ID) enthält.

Wie bereits in Kapitel 2.2 erwähnt wurde, sollten Datenbanktabellen in Normalformen gebracht werden. Dabei muss jedes Datenfeld aus einem atomaren Wert bestehen, den man nicht weiter zerlegen kann. Außerdem sollten keine Informationen mehrfach, also redundant, vorkommen, da man ansonsten Probleme mit der Aktualisierung und Löschung von Daten bekommt. Wird eine Aktualisierung nicht vollständig ausgeführt, weil beispielsweise die Netzwerkverbindung gestört ist, ist die Datenbank in einem inkonsistenten Zustand. Der gesamte Datenstamm kann dadurch unbrauchbar werden,

da der „richtige“ Zustand einer inkonsistenten Datenbank meist nicht mehr ermittelt werden kann. Um dies zu vermeiden, muss neben einer korrekten ER-Modellierung das in Kapitel 2.2 vorgestellte Prinzip der Transaktionen zum Einsatz kommen.

Bereits 1976 stellte Peter Chen eine Notation vor, mit der man die ER-Modellierung durchführen kann. Solche Modelle müssen Sie als (Datenbank-)Entwickler in Kooperation mit dem Kunden erstellen. Die so genannte Chen-Notation bildet dabei die Gesprächsgrundlage in der Phase der Anforderungsanalyse, die letztlich eine korrekte Modellierung ermöglichen soll.

Abbildung 3.7 skizziert ein ER-Diagramm, das aus drei Entitäten (= Datenbanktabellen) besteht, nämlich Autor, Buch und Verlag. Diese Entitäten stehen in Relation zueinander, die zunächst textuell beschrieben wird. Jede Relation beinhaltet eine Kardinalität (1:1, 1:n oder n:m), wobei die mehrfachen Bezüge n und m durch * ausgedrückt werden. Im Beispiel kann ein Autor mehrere Bücher schreiben, an einem Buch können auch mehrere Autoren beteiligt sein. Es handelt sich also um eine n:m-Beziehung. Ein Verlag vertreibt mehrere Bücher, ein Buch wird jedoch immer genau von einem Verlag verbreitet. Hier handelt es sich also um eine 1:n-Beziehung. Zusätzlich sind noch die wichtigsten Felder jeder Tabelle dargestellt. So besitzt ein Buch beispielsweise eine ISBN-Nummer und einen Titel.

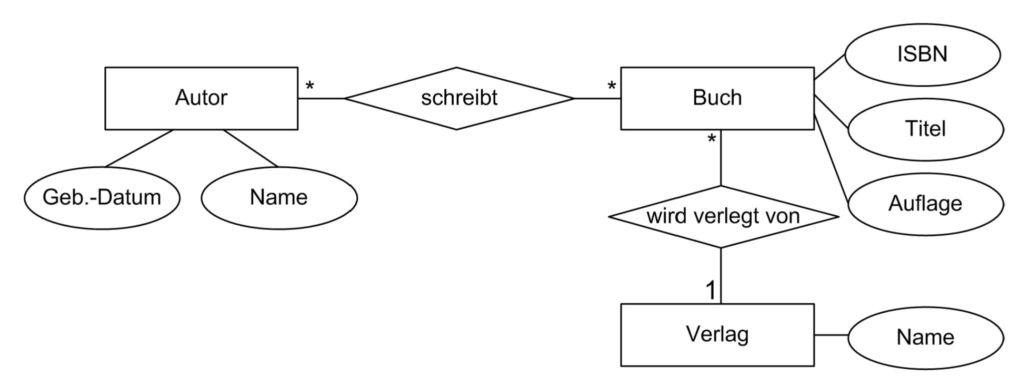


Abbildung 3.7: Skizze eines ER-Modells

Wenn Sie diese Tabellen über PHPMyAdmin auf dem MySQL-Server angelegt haben, können Sie über die in Kapitel 2.2 beschriebenen PHP-Befehle Funktionen zum lesenden und schreibenden Zugriff auf diese Daten erstellen und als einzubindende *.inc.php*-Dateien für die Fachlogik anbieten.

Die Fachlogik

Die Fachlogik wird in enger Abstimmung mit dem Kunden erstellt, da sie seinen Geschäftsprozess repräsentiert, der in Zukunft mit der PHP-Anwendung realisiert werden soll. Im Wasserfallmodell sind die geforderten Dienste/Funktionalitäten im Lastenheft vom Kunden definiert und aus Sicht des Systemanalytikers und Entwicklers im Pflichtenheft genauer spezifiziert worden. Die Fachlogik besteht aus PHP-Dateien, die nahezu ausschließlich PHP-Code (also weder SQL-Anweisungen noch HTML-, JavaScript- oder gar CSS-Befehle) enthält. Sie beinhaltet die „Intelligenz“ der Anwendung, die auf dem Web-

server ausgeführt wird und realisiert die Dienste der Artikel-, Kunden- oder Bestellverwaltung. Abbildung 3.6 skizziert die Dienste „Kunde anlegen“ und „Kunde suchen“, die in einer *kundenverwaltung.php* als Funktionen realisiert sein können.

Bei einem Vorgehen nach dem Spiralmodell werden die benötigten Dienste in den Anforderungen an die Software vom Kunden ermittelt und im Entwicklungsplan festgehalten (Abb. 3.2). Die Spezifikation der Entwickler erfolgt im Produktentwurf. Durch die frühe Erstellung von Prototypen der Anwendung kann der Kunde jedoch stets Einfluss auf die Implementierung nehmen.

Beim V-Modell gehört die Spezifikation der Dienste der Fachlogik in die Phase der Anforderungsstellung an die Entwickler, die auf dieser Basis einen Entwurf des Systems vornehmen. Dabei werden die Module wie Kundenverwaltung und Artikelverwaltung mit ihren Diensten zunächst formal in Textform oder in Ablaufdiagrammen beschrieben. Diese bilden die Anforderungen an die Komponenten. Dabei sollten auch bereits Grenzwerte für Eingaben betrachtet werden, z. B. mögliche Rabattstufen auf einen Artikel. In der Phase des Komponentenentwurfs werden die Dienste der Fachlogik dann zusammen mit den Diensten der Datenzugriffs- sowie der Präsentationsschicht entwickelt. Im Anschluss daran erfolgen zunächst ein Test der einzelnen Dienste (z. B. des Anlegens eines Neukunden) und nachfolgend die Integration der Dienste in die zu testende Gesamtanwendung.

Die Präsentationsschicht

Die Präsentationsschicht einer PHP-Anwendung ist dominiert von der Ausgabe des PHP-Interpreters, die zum Internetbrowser des Clients gesendet wird. Im Kontext des Webs 2.0 muss dies keine reine Ausgabe von dynamisch erzeugtem HTML-Code sein. Vielmehr steht einem modernen Internetbrowser wie dem Microsoft Internet Explorer 8.0 und dem Mozilla Firefox 3.x eine Vielzahl von Technologien zur Verfügung, die eine Interaktion der Anwendung mit dem Benutzer ermöglichen. Dazu gehören

- dynamisch generierter HTML-Code, wie Tabellen oder HTML-Formulare, die vom Benutzer auszufüllen sind. Die ausgefüllten Formulare werden dann vom Server interpretiert. Die verbesserten Möglichkeiten von CSS2 (Cascading Style Sheets) sorgen hier für ein optisches Erscheinungsbild, das eher an eine lokale Anwendung als an ein tristes Internetformular erinnert.
- der Versand oder die dynamische Erstellung von PDF-Dateien und PDF-Formularen, die ebenfalls ausgefüllt und per PHP serverseitig interpretiert werden können.
- der Funktionsumfang der AJAX-Programmierung (Asynchronous JavaScript and XML). Die neuen Möglichkeiten der JavaScript-Programmierung mit einer clientseitigen JavaScript Engine sorgen dafür, dass Webseiten nicht bei jedem Aufruf vollständig neu geladen werden müssen. Serverseitig kann hier durchaus auch PHP zum Einsatz kommen.
- das Versenden von Adobe-Flash- oder Microsoft-SilverLight-Quellcode, deren Aufrufe in HTML-Quellcode eingebunden werden. Flash und SilverLight verfügen über eine Vielzahl von optischen Gestaltungseffekten und hoher Multimedialität, sodass ansprechende browserbasierte Anwendungen erstellt werden können. Da PHP den HTML-Code generiert, kann die Skriptsprache somit auch Einfluss auf die Parametrisierung der übertragenen Fremdformate nehmen.

- der Versand von Java-Applets, die über den Applet-Tag des HTML-Befehlssatzes mit zum Client übertragen werden können. Die Applet-Technologie gilt jedoch als veraltet.

Neben der Technologie besteht das Ziel eines Entwicklers der Präsentationsschicht darin, eine für den Benutzer angenehm bedienbare Schnittstelle zu bieten (Stichwort: Softwareergonomie), die in alltägliche Geschäftsvorgänge des Benutzers integriert ist.

Stellen Sie sich vor, der Bediener möchte beispielsweise einen neuen Kunden erfassen. Dazu sind die folgenden Daten in die Datenbank einzuspeisen:

- Name und Vorname
- ggf. Firmenname
- Anschrift mit Straße, Hausnummer, PLZ und Ort
- ggf. Telefonnummer
- ggf. E-Mail-Adresse

Zwingt die Präsentationsschicht den Bediener, diese Daten in einer anderen Reihenfolge als die oben genannte einzugeben, so wird der Bediener die Anwendung als störend empfinden, da die Eingabe nicht dem natürlichen Erfassen der Daten aus dem Alltag entspricht.

Zusätzlich dazu muss die Präsentationsschicht mit ihrem Design und der Farbwahl in die Corporate Identity des Unternehmens eingebunden sein. Dies gilt insbesondere für PHP-Anwendungen, die direkt dem Endkunden zur Verfügung gestellt werden, wie es bei B2C-, B2B- oder B2E-Portalen der Fall ist.

Des Weiteren kann bereits in der Präsentationsschicht eine Vorgabe für das Format einer Eingabe festgelegt werden. Diese wird dann nach der Übergabe an die Fachlogik auf ihre Gültigkeit geprüft. Bei der Verwendung von Freitextfeldern hat der Entwickler der Präsentationsschicht die wenigsten Probleme, die Prüfung der Eingabe und die einheitliche Ablage in der Datenbank erfordert jedoch einen stark erhöhten Aufwand. Im Folgenden werden einige Möglichkeiten zur Eingabe einer Telefonnummer vorgestellt:

- 016098369800
- 0160-98369800
- 0160/98369800
- +49(0)160-98369800
- 004916098369800

Wenn verschiedene Mitarbeiter Telefonnummern anlegen können, werden Sie bei einer Freitexteingabe all diese Formate in der Datenbank finden. Eine Möglichkeit der Präsentationsschicht besteht darin, nur genau eine dieser Formen zuzulassen.

Ob eine Benutzerschnittstelle gelungen ist oder nicht, wird im Wasserfallmodell in der Testphase und beim V-Modell beim Abnahmetest festgestellt. In der Realität ist dies zu spät, da das Projekt bereits viel zu weit fortgeschritten und der Zeitdruck zu groß ist, um Änderungen vorzunehmen. Das Resultat besteht in einer mangelnden Akzeptanz der Anwender. Das Spiralmodell erwähnt bereits den frühen Entwurf von Prototypen, die

sinnvollerweise auch den Endanwendern zur Verfügung gestellt werden sollten. Ein früher GUI-Prototyp als Diskussionsgrundlage kann hier bereits in einer frühen Projektphase Designentscheidungen hervorrufen, die die Akzeptanz fördern.

Die Arten der Prototypen werden im folgenden Kapitel vorgestellt. In Kapitel 3.2 werden im Kontext der Objektorientierung agile Vorgehensweisen skizziert, die die Kommunikation zu allen Projektbeteiligten stärker fokussieren mit dem Ziel einer Erhöhung der Akzeptanz der zukünftigen Anwendung.

Arten des Prototypings

Bei der Erstellung einer Anwendung ist es selbstverständlich geworden, dem Auftraggeber bereits vor der Abgabe eine nicht vollständig fertige Vorversion zu präsentieren. Diese wird Prototyp genannt. Man hat aus den bösen Überraschungen bei der Wasserfall-Vorgehensweise gelernt, bei der die Auftraggeber und auch die Anwender die Anwendung erst bei der Inbetriebnahme zu Gesicht bekommen. Selbst bei kleineren Projekten kommt es dabei kaum vor, dass die Abnahme problemlos verläuft und die Auftraggeber und Anwender mit der ersten Version vollständig zufrieden sind. Wie bereits erwähnt, sind Änderungswünsche umso schwieriger umzusetzen, je mehr die individuelle Anwendung bereits fertig gestellt und je weiter das Projekt fortgeschritten ist, da ein Großteil der Ressourcen (Arbeitszeit und Budget) bereits aufgebraucht sind.

Einem (freiberuflichen) Programmierer, der zunächst eine Programmiersprache lernt und sich im Verlauf seiner Projekte Methoden des Projektmanagements erst aneignet, sind zumeist die verschiedenen Arten des Prototypings in der Softwareentwicklung nicht geläufig. Denn auch das Prototyping sollte strukturiert erfolgen und sich einem Ziel widmen. Dazu existieren verschiedene Verfahren, wie man einen Prototyp entwickeln kann.

Ein weit verbreiteter Prototyp in der Softwareentwicklung ist der vertikale Prototyp. Hier wird nur eine Ebene des Gesamtsystems exemplarisch realisiert. Dies ist meist die Präsentationsschicht, sodass ein GUI-Prototyp entsteht. Der Kunde erhält dadurch einen Einblick in das zukünftige Design der PHP-Anwendung, denn in den Eingabefeldern können bereits Testdaten eingegeben werden. Auch den Ablauf eines Geschäftsprozesses, wie die Erstellung eines Neukunden, kann man ausprobieren. Dazu sind mehrere Formulare z. B. über HTTP-POST-Schaltflächen miteinander verknüpft. Die Daten selbst werden jedoch noch nicht von einer Fachlogik analysiert, aufbereitet und abgelegt. Ziel eines solchen Prototyps, der in Abbildung 3.8 skizziert wird, ist die Präsentation für die Auftraggeber und die zukünftigen Anwender. Der Prototyp wird als Diskussionsgrundlage für das Design und den Ablauf der Geschäftsprozessmodellierung im System gesehen. Ein frühes Bekanntmachen der neuen Anwendung erhöht ebenso die Akzeptanz bei den Benutzern. Dies gilt insbesondere dann, wenn die Benutzer durch eigene Vorschläge aktiv an dem Gestaltungsprozess teilhaben können. Dieses Prototyping wurde erstmals im Spiralmodell der Softwareentwicklung spezifiziert.

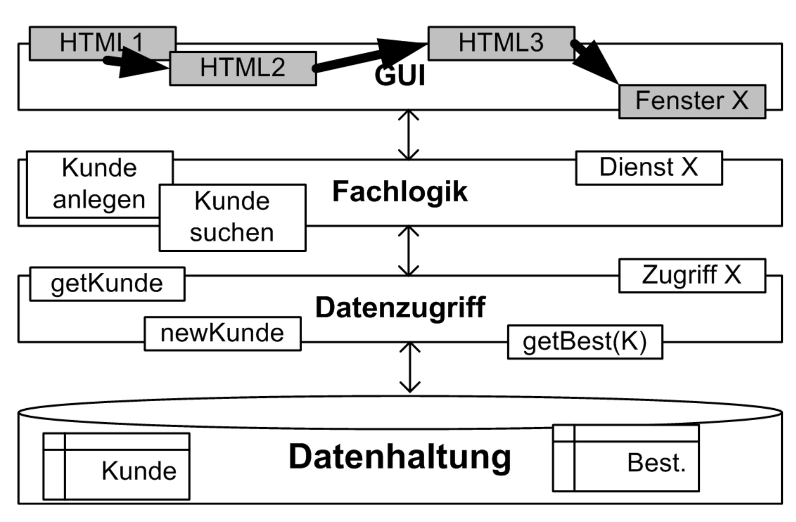


Abbildung 3.8: Ein horizontaler Prototyp

Eine ganz andere Zielsetzung hat ein horizontaler Prototyp. Sie wollen unter anderem ein Eingabeformular erstellen, bei dem Sie einen Neukunden erfassen. Dies ist nur einer von vielen Diensten, die Sie im Rahmen Ihres Projekts realisieren sollen. Sie haben aber bislang noch nie mit PHP gearbeitet. Außerdem kennen Sie prinzipiell den Zugriff auf eine MySQL-Datenbank, haben einen solchen Zugriff aber noch nie realisiert. Die Idee, einen einzelnen Dienst durch alle Schichten hindurch zu implementieren, wird in einem horizontalen Prototyp umgesetzt. Dieser ist für den Kunden zunächst uninteressant, denn er sieht lediglich die eine Eingabemaske. Viel wichtiger ist ein solcher Prototyp für Sie als Entwickler, da mit ihm noch vorhandene Funktionalitäts- oder Implementierungsfragen geklärt werden können. Nach der Umsetzung des Prototyps können Sie also Testkunden im System anlegen und sich vergewissern, dass dies mit der von Ihnen vorgeschlagenen WAMP- oder LAMP-Architektur auch funktioniert.

Beispiel

Implementieren Sie einen Dienst zum Anlegen eines Testkunden, indem Sie zunächst eine entsprechende Datenbanktabelle in MySQL über PHPMyAdmin anlegen. Im zweiten Schritt entwerfen Sie eine PHP-Funktion für die Fachlogik mit Namen *anlegenKunde*, die die Parameter für einen Neukunden als Input entgegennimmt, eine Verbindung zum Datenbankserver aufbaut, die Daten ablegt und die Verbindung wieder schließt. Erstellen Sie im dritten Schritt ein passendes HTML-Formular zur Eingabe der Daten, das auf eine PHP-Seite verweist, die die Daten dann ablegt und über den Erfolg als HTML-Ausgabe berichtet. Die entsprechenden PHP-Funktionen sind in Kapitel 2.2 beschrieben.

Neben dem Test der Technologie kann mit einem vertikalen Prototyp auch die Performance der PHP-Anwendung auf dem Server getestet werden. So können Sie beispielsweise nahezu gleichzeitig von verschiedenen Clients aus 100 oder 1 000 Kunden anlegen lassen, indem HTTP-Anfragen mit ausgefüllten Formulardaten an den Webserver automa-

tisiert abgesendet werden. Der Lasttest zeigt, ob alle Clients in einer angemessenen Antwortzeit eine Erfolgsmeldung der Erstellung erhalten haben oder nicht. Ein Anwender wartet nicht gerne länger als 1-2 Sekunden auf die Antwort vom System, nachdem er eine Eingabe getätigt hat. Eine längere Wartezeit führt zu Frustration über das langsame System und damit zu verminderter Akzeptanz der Anwender. Andererseits lässt sich mit dem Lasttest aber auch prüfen, ob alle Kunden korrekt angelegt wurden. Besitzt wirklich jeder Kunde eine eindeutige ID oder wurden Kundennummern doppelt vergeben?

Abbildung 3.9 skizziert den Weg durch die Schichten bei einem vertikalen Prototyp.

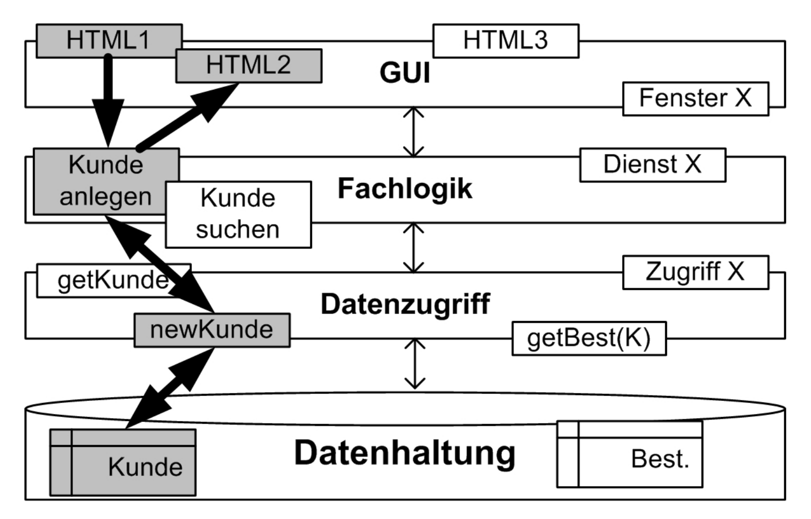


Abbildung 3.9: Ein vertikaler Prototyp

Eine weitere Unterscheidung der Prototypen liegt in ihrer Wiederverwendung. Beim *Rapid Prototyping* erstellen Sie einen Prototyp, der nur zur Erkenntnisgewinnung dient, beispielsweise um eine der folgenden Fragen zu beantworten:

- Wünscht sich der Kunde diese Art der Navigation?
- Funktioniert dieser Dienst mit diesem WAMP-Server prinzipiell?
- Wie performant ist die Anwendung?

Das Rapid Prototyping dient also zu Forschungszwecken bzw. für die Suche nach Möglichkeiten zur Realisierung einer Problemlösung. Die gewonnenen Erkenntnisse können anschließend für das richtige Produkt weiterverwertet werden, indem Sie mit den Erkenntnissen eine umfangreiche Problemanalyse und Systemspezifikation durchführen. Der Quellcode des Prototyps wird jedoch nicht selbst zum Produkt ausgebaut. Man spricht hier von einem „Wegwerf-Prototypen".

Bei einem evolutionären Prototyping ist dies anders. Hier werden die Funktionalität und damit der Quellcode des Prototyps schrittweise erweitert, bis sich daraus das endgültige Produkt formt. Die Erweiterungen werden anhand des Feedbacks der zukünftigen Anwender bzw. des Auftraggebers vorgenommen. Der Prototyp wird dabei stets lauffähig gehalten und bis zur Produktreife weiterentwickelt. Auf den ersten Blick scheint das

evolutionäre Prototyping aufgrund der zusätzlichen Wiederverwendung des Codes wirtschaftlicher zu sein als das Rapid Prototyping. Neue Erkenntnisse gewinnt man ja in beiden Fällen.

Falls Sie jedoch schon einmal Software entwickelt und erfolgreich fertiggestellt haben, haben Sie sich sicherlich Folgendes gesagt: „Ich bin froh, dass es funktioniert, aber beim nächsten Mal würde ich alles anders/besser machen!“ Beim evolutionären Prototyping neigt man nämlich dazu, funktionierenden (schlechten) Quellcode beizubehalten, da die Änderung Zeit kostet und neue Probleme mit sich bringen kann.

Um das Gesamtdesign der evolutionären Anwendung robust zu halten und die Gefahr des „gehackten“ Quellcodes zu verringern, müssen Sie regelmäßig Refactorings durchführen. Damit ist eine Strukturverbesserung des Quellcodes unter Beibehaltung des Verhaltens der Anwendung gemeint. Ein Refactoring hat das Ziel, die Lesbarkeit, Verständlichkeit, Wartbarkeit und damit auch die Erweiterbarkeit der gesamten Anwendung zu verbessern. Wenn Sie also eine evolutionäre Entwicklung Ihrer PHP-Anwendung planen, sollten Sie sowohl in der Zeit-, als auch in der Kostenplanung regelmäßige Refactorings berücksichtigen.

3.2 Objektorientierte Programmierung

Mit der Verbreitung der Programmiersprache Java in den letzten 10 Jahren hat sich das Programmierparadigma der Objektorientierung verbreitet. Als passende Beschreibungssprache, die die Ideen der Objektorientierung beinhaltet, hat sich nahezu zeitgleich die Unified Modeling Language (UML) etabliert. Java hat sich insbesondere serverseitig mit dem Konzept der Enterprise Java Beans (EJB) in der aktuellen dritten Version sowie der Skriptsprache JSP (Java Server Pages) und Servlets durchgesetzt. Dem hat Microsoft das .NET-Framework mit den Sprachen C#, VisualBasic.NET und ASP.NET (Active Server Pages) ebenfalls ausschließlich mit objektorientierten Konzepten gegenübergestellt. JSP im Java- und ASP im Microsoft-Umfeld kann man als konkurrierende Lösung zu PHP-Anwendungen ansehen. Während die Objektorientierung in PHP4 nur rudimentär unterstützt wurde, kann man mit PHP5 die Konzepte der Objektorientierung vollständig umsetzen.

In diesem Kapitel wird nun die objektorientierte Denkweise mit ihren Ideen und Techniken zunächst unabhängig von der Sprache PHP vorgestellt. Die Umsetzung in PHP wird im vierten Kapitel dieses Buches präsentiert.

3.2.1 Typische Projektgröße und Projektdefinition

Zunächst einmal ist die Frage zu stellen, warum es sich bei der Objektorientierung um ein neues Paradigma – also um ein grundlegend neues Prinzip – der Anwendungsentwicklung handelt. Die Objektorientierung erhebt den Anspruch, menschliche Organisationsmethoden aus der realen Welt besser nachzubilden als die bislang vorgestellten Konzepte der prozeduralen und modularen Programmierung. Während diese Konzepte die Denkweise der Maschinen mit sequenziellen Anweisungen, Unterprogrammaufrufen, Rücksprüngen zu aufrufenden Methoden, einer Teilung von GUI, Fachlogik und

Datenzugriff mit unterliegender, zumeist relationaler Datenbankstruktur in den Vordergrund stellten, liegt der Fokus der Objektorientierung zunächst auf den Fragestellungen:

1. Was soll die Anwendung leisten, welche Funktionalität soll sie besitzen?
2. Wie ist der Ablauf der Geschäftsprozesse, die abgebildet werden sollen?
3. Was soll überhaupt modelliert werden?

Die objektorientierte Denkweise richtet sich also stärker an den Kunden mit seinen Anforderungen an die zu erstellende Anwendung als bei einer prozeduralen oder modularen Vorgehensweise. Die technischen Details interessieren den Kunden in Wirklichkeit wenig; sogar ob PHP zum Einsatz kommt oder eine andere Sprache:

Ein kleiner Händler, der nebenbei einen Online-Shop mit 10 Artikeln verwalten will, besitzt meist nicht die Kenntnis von einer 3-Tier-Infrastruktur oder von einem PHP-Interpreter. Von einem guten Programmierer wird diese Selbstverständlichkeit meist nicht wahrgenommen. Stattdessen will er oft seinem Kunden stolz die Funktion seiner neuen Anwendung detailliert erklären. Dies interessiert den Kunden jedoch nicht. Er möchte lediglich seinen Shop mit den gewünschten Funktionen möglichst leicht handhabbar online stellen.

Die Projektbeteiligten

Die Methoden der Objektorientierung und der UML sind vor allem dann sinnvoll anwendbar, wenn an dem Projekt der Softwareerstellung eine Vielzahl von Personen beteiligt ist. Nur bei kleinen Projekten haben Sie als Entwickler direkt und ausschließlich Kontakt zu einem (einzelnen) Verantwortlichen auf der Seite des Kunden. Bei größeren (PHP-)Projekten spielen jedoch viel mehr Personen eine Rolle; im positiven wie im negativen Sinne.

Alle Personen, die direkt oder indirekt Einfluss auf ein Softwareprojekt haben, werden als Projektbeteiligte oder Stakeholder bezeichnet. Dabei kann es sich um natürliche Personen, eine Personengruppe oder auch um Institutionen handeln. Es werden drei Arten von Stakeholdern unterschieden, die nach ihrem Einfluss- und Wirkungsgrad abgestuft sind. Der Begriff des Wirkungsgrads stammt ursprünglich aus der Physik und beschreibt das Verhältnis von abgegebener Leistung bzw. Nutzen zu dem zugeführten Aufwand. Diese Definition lässt sich unbedenklich auf Projekte übertragen. Man unterscheidet

- primäre Stakeholder, die einen hohen Einflussgrad auf das Projekt haben, jedoch nur einen geringen Wirkungsgrad besitzen. Sie sind in die Hauptinteraktion mit dem Produkt involviert, wie Entwickler oder Anwender.
- sekundäre Stakeholder, die einen niedrigen Einflussgrad und gleichzeitig einen geringen Wirkungsgrad auf das Projekt besitzen. Sie sind nicht direkt beteiligt, haben aber ein Interesse an dem Produkt oder besitzen eine vermittelnde Rolle bei den Entwicklungsaktivitäten. Dies können externe Berater sein.
- Key-Stakeholder mit einem geringen oder hohen Einflussgrad, jedoch auf jeden Fall einem hohen Wirkungsgrad auf das Projekt. Sie haben entscheidenden Einfluss bei der Produktdefinition und sind bedeutend für den Erfolg der Entwicklungsaktivitäten. Dabei handelt es sich meist um Entwicklungsleiter, Geld- und/oder Auftragsgeber.

Für einen erfolgreichen Projektverlauf ist es wichtig, dass Sie die einzelnen Stakeholder-Gruppen korrekt identifizieren und entsprechend mit ihnen kommunizieren. Abbildung 3.10 fasst die beteiligten Rollen zusammen.

Bei großen Projekten arbeiten Sie als PHP-Entwickler normalerweise nicht allein als Programmierer, vielmehr sind die Aufgaben in einem Entwicklerteam aufgeteilt. Jeder Entwickler ist für eine Reihe von zu entwickelnden Komponenten zuständig. Die Komponenten sind dann zusammenzufügen und bilden die Anwendung. Bereits an dieser Stelle treten oft Probleme auf, falls die Schnittstellen der Komponenten zueinander nicht klar festgelegt wurden oder zu dem Zeitpunkt der Festlegung noch gar keine Spezifikation möglich war.

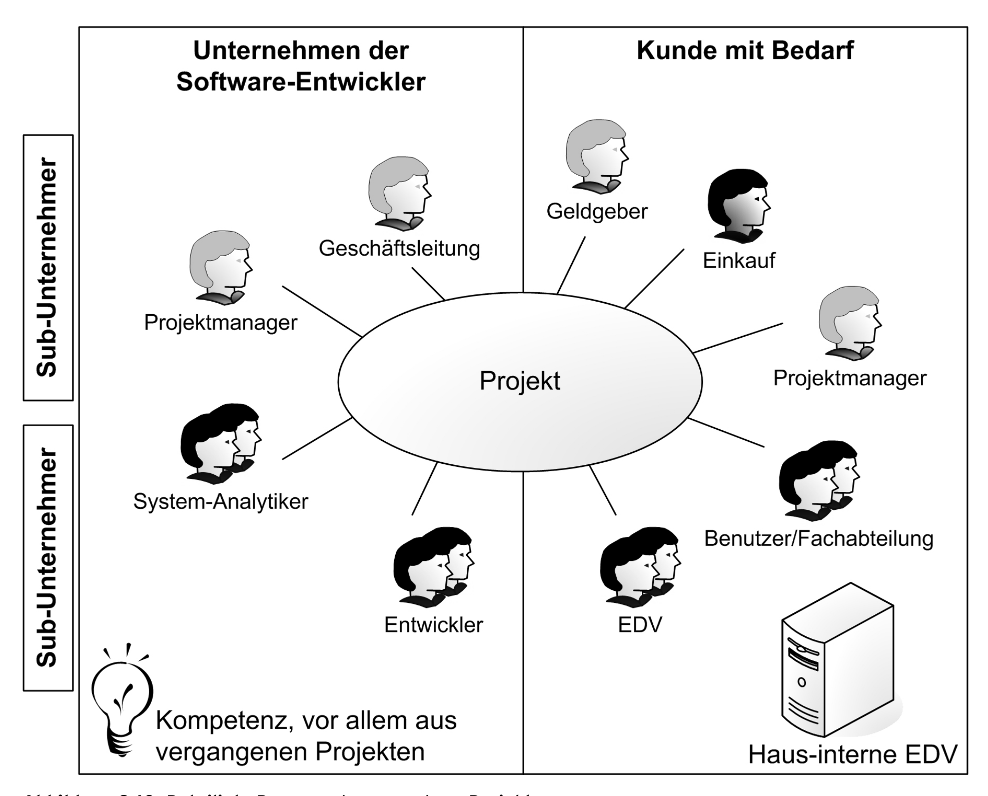


Abbildung 3.10: Beteiligte Personen(-gruppen) am Projekt

Bei größeren Projekten beginnen Sie als Entwickler nicht sofort bei Projektbeginn mit ihrer Arbeit. Ihrer Tätigkeit ist die Tätigkeit einer oder mehrerer Systemanalytiker vorgeschaltet. Der Begriff „vorgeschaltet" darf jedoch nicht so verstanden werden, dass die Analytiker ihre Arbeit beenden, bevor Sie beginnen. Dies wäre eine Wasserfallmethode. Vielmehr sollen Systemanalytiker Ihnen als Entwickler bei der Kommunikation mit dem Kunden helfen, der üblicherweise aus einer völlig anderen Fachdomäne – vielleicht aus dem Banken-, Versicherungs- oder Gesundheitswesen – stammt. Man muss davon ausgehen, dass ein Versicherungsmakler oder ein Arzt als zukünftiger Benutzer aus der Fachabteilung keinerlei Kenntnisse eines Entwicklers besitzt.

Auf der Seite Ihres Unternehmens ist zusätzlich das Projektmanagement zu nennen, das den Zeit- und Kostenplan und die gesamten Ressourcen der Entwicklung überwacht. Das Projektmanagement kann dabei in direkter Kommunikation zu Ihrer Geschäftsleitung stehen, insbesondere bei größeren Vorhaben und hohem Auftragsvolumen. Dies sind demnach die Stakeholder Ihres Unternehmens. Falls Ihr Unternehmen den Auftrag gar nicht vollständig selbst umsetzt, kommen als zusätzliche Stakeholder Ihre Subunternehmer hinzu. Das Outsourcing der Programmierung sogar in andere Länder ist bei größeren Konzernen durchaus üblich. Ebenso wird die Systemanalyse unter Umständen von einem externen Consulting-Unternehmen durchgeführt, das entweder von Ihrer Firma, oder vom Kunden beauftragt wurde. Zusätzlich besitzen Sie in Ihrem Unternehmen eine gewisse fachliche und technische Kompetenz aufgrund der Ausbildung aller Mitarbeiter und der gesammelten Erfahrung aus vergangenen Projekten.

Auch auf der Seite des Kunden existiert neben der bereits angesprochenen Gruppe der Anwender aus der Fachabteilung zumeist ein Projektmanager, der jedoch oft selbst kein Benutzer der Software ist. In manchen Fällen stammt der Projektmanager aus der EDV-Abteilung des Kunden, er kann jedoch auch ein Führungsmitglied der Fachebene – also der jeweiligen Fachdomäne – sein. Üblich ist auch, dass bei Ihrem Kunden eine eigene EDV-Infrastruktur existiert, in die Ihre Anwendung einzugliedern ist. Es ist nicht selbstverständlich, dass in größeren Unternehmen ein externer Provider Ihre PHP-Anwendung hostet. Ein entsprechender WAMP- oder LAMP-Server kann auch vom Kunden selbst betrieben und in seine eigene Systemlandschaft, beispielsweise zu einem SAP-System, integriert werden.

Existiert ein Projektmanager, so dürfen die Anwender, die die Software letztlich bedienen, als primäre Stakeholder auch in frühen Projektphasen nicht vergessen werden. Auf höherer Managementebene existiert beim Kunden meist noch separat ein Geldgeber, dem der Projektmanager des Kunden fachlich untergeordnet ist. Die Abteilung des Einkaufs auf der Kundenseite muss von Ihrem Unternehmen davon überzeugt werden, die Software in Ihrem Hause entwickeln zu lassen.

Eine Vielzahl an Interessen

Festzuhalten ist, dass sowohl auf der Seite Ihres eigenen Unternehmens als auch auf der des Kunden insbesondere bei großen Projekten eine Vielzahl von Projektbeteiligten eine Rolle spielen, die jeweils ihren eigenen fachlichen und/oder technischen Hintergrund besitzen und in der Regel neben dem Erfolg des Projekts auch eigene Interessen verfolgen.

Im Folgenden werden die typischen Interessen der Stakeholder auf der Seite Ihres Kunden kurz skizziert:

- Geldgeber/Geschäftsleitung:
  - Einhaltung von unternehmensweiten Standards
  - Zuschnitt der neuen Anwendung auf die Bedürfnisse des eigenen Unternehmens
  - lange Gewährleistung und Verfügbarkeit der Entwickler bei Problemen
- Einkauf:
  - Erhalt einer preisgünstigen Lösung

- Anwender:
  - unkomplizierte Handhabung der Anwendung
  - alle gewünschten Funktionen sind vorhanden
  - Anpassung der Anwendung an die Abläufe im Arbeitsalltag
- EDV-Abteilung:
  - Integration der neuen Anwendung in die Systemlandschaft
  - Stabilität der Anwendung
  - leichte, zentrale Wartung
  - langfristige Anpassung an neue Gegebenheiten

Alle diese Benutzergruppen sollen durch Ihre Anwendung zufriedengestellt werden. Einige der Zielsetzungen sind sogar konträr zueinander, unter anderem, da die Wünsche der Geschäftsleitung, Anwender und der EDV-Abteilung meist nicht in einer preisgünstigen Lösung realisierbar sind. Die in Kapitel 3.2.3 vorgestellten agilen Methoden sollen Ihnen dabei helfen, die Wünsche der einzelnen Zielgruppen zu erfassen.

Große Projekte mit Objektorientierung

Während das Wasserfallmodell in der strikten Form mit Lasten- und Pflichtenheft sowie mit den im Vorfeld geplanten Abgaben in der Praxis nur bei kleinen Projekten bis zu 2 Mannjahren erfolgreich war, ist eine Vorgehensweise nach dem Spiral- oder V-Modell bereits bei größeren Projekten erfolgversprechend.

Die Vorteile eines objektorientieren Ansatzes kommen insbesondere bei sehr großen Projekten ab 200 Personenjahren zur Geltung, die nicht mehr von einer einzelnen Person verwaltet, geschweige denn realisiert werden können. Bei solchen Projekten werden oft einige Millionen Zeilen an Quellcode produziert. Bei dieser Projektgröße kommt auch nicht mehr ein einzelner Entwickler zum Einsatz, sondern ein ganzes Team von Entwicklern. Ihr Arbeitgeber kann sich den Ausfall des Stammentwicklers durch Krankheit oder Verlassen des Unternehmens nicht leisten. Außerdem würde die Implementierung viel zu viel Zeit benötigen, sodass sich die Anforderungen unter Umständen schneller ändern als sie implementiert werden können. Das Entwicklerteam benötigt wiederum ein eigenes Management zur Planung der Kapazitäten und Aufgaben, wobei das Management nicht zwangsläufig denselben Hintergrund wie die Entwickler besitzt, da es meist aus einer anderen fachlichen Domäne stammt.

Bei einem Projekt dieser Größenordnung muss eine projektbezogene Kommunikation der in Abbildung 3.10 dargestellten Stakeholder erfolgreich möglich sein. Neben technischen Problemen werden die verschiedenen Fachbereiche der Beteiligten zu einem größeren Problem, dem man mit sozialkommunikativen Fähigkeiten – so genannten „Soft Skills" – und einer gemeinsamen Sprache entgegenwirken muss. Die Objektorientierung bietet mit der UML eine Notation, die von allen Beteiligten in ihrem jeweiligen Wirkungskreis leicht verstanden und als Diskussionsgrundlage verwendet werden kann.

Dass große Projekte mit PHP und MySQL realisierbar sind, zeigt unter anderem das unter der GPL (General Public License) stehende Content-Management-System Typo3 (*http://typo3.org/*).

Projekte dieser Größenordnung sind dynamisch, nicht vollständig im Vorfeld planbar und erfordern dennoch eine wohldefinierte Vorgehensweise. Während die bislang vorgestellten Methoden und Modelle eher statisch orientiert waren, beinhaltet die Objektorientierung mit agilen Methoden eine größere Flexibilität, um sich Änderungen der Anforderungen anpassen und neue, noch unbekannte Anforderungen in dem laufenden Projekt einbeziehen zu können.

Es wurde bereits erwähnt, dass die Objektorientierung den Anspruch erhebt, eher an der Denkweise des Menschen ausgerichtet zu sein. Es hat sich gezeigt, dass Verwaltungssysteme aller Art relativ leicht objektorientiert modelliert werden können. Dies können beispielsweise sein:

- eine Kundenverwaltung
- eine Artikelverwaltung
- eine Auftragsverwaltung
- eine Aktienverwaltung
- eine Verwaltung von Musik und Musikmedien

Wenn Sie einige Verwaltungssysteme modelliert haben, wird Ihnen auffallen, dass die Struktur dieser Systeme stets sehr ähnlich ist. So ist es in der Objektorientierung nur von untergeordneter Bedeutung, ob Sie in einer Artikelverwaltung nun Autos oder Bücher verwalten.

Nach einer Definition der Begriffe der Objektorientierung werden insbesondere im vierten Kapitel dieses Buches diese Verwaltungssysteme skizziert. Eine gute Übung wird darin bestehen, diese Beispiele im ersten Schritt nachzuvollziehen und zu verstehen, im zweiten Schritt selbst nachzuprogrammieren und zu erweitern.

Der Rational Unified Process (RUP)

Bevor in die Begriffe der Objektorientierung eingestiegen wird, soll zunächst das passende Vorgehensmodell kurz vorgestellt werden. Der Rational Unified Process (RUP) ist ein objektorientiertes Vorgehensmodell zur Softwareentwicklung. Es wurde 1997 als kommerzielles Produkt der Firma Rational Inc. entwickelt und liegt mittlerweile in der neunten Version vor.

Interessant ist dabei, wer das Vorgehensmodell entwickelt hat. Es handelt sich um die Programmierer Grady Booch, Ivar Jacobson und James Rumbaugh, die in dieser Zeit bei Rational Inc. angestellt waren. Diese drei Entwickler haben seit 1995 ebenfalls die Syntax der UML erstellt und gelten als „Väter“ der UML. Die Standardisierung und Weiterentwicklung der UML wurde an die Object Management Group (OMG) übergeben, die dann im Januar 1997 offiziell die erste Version der UML herausbrachte. Das Modell des Rational Unified Process benutzt seinerseits die UML als Notationssprache. Abbildung 3.11 zeigt das zentrale Schaubild des RUP.

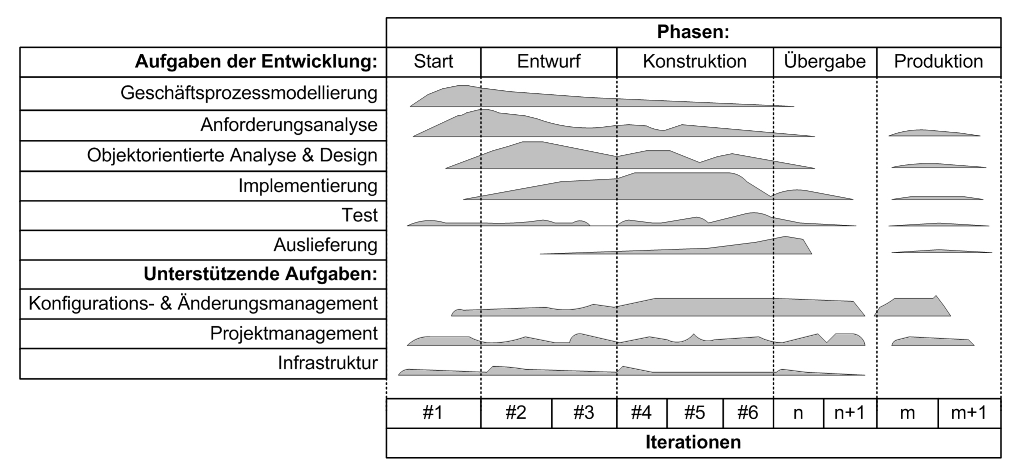


Abbildung 3.11: Der Rational Unified Process

Der RUP gilt als letztes schwergewichtiges Vorgehensmodell der Softwareentwicklung. Ein schwergewichtiges Modell ist dadurch gekennzeichnet, alle Anforderungen an die zu entwickelnde Anwendung in einer Projektphase vollständig zu erheben, bevor die ersten Entwurfs- oder Realisierungsentscheidungen getroffen werden. Abbildung 3.11 zeigt, dass die Anforderungsanalyse im RUP jedoch bereits deutlich aufgeweicht ist.

Prinzipiell unterscheidet der RUP den Projektstart, die Phase des Entwurfs der Software, die Konstruktion, die Übergabe an den Kunden sowie die Produktion, in der die Software im operativen Einsatz des Kunden ist. Die zeitlichen Abläufe sind jedoch nicht klar voneinander abgegrenzt.

Quasi auf der Y-Achse definiert der RUP die zu erfüllenden Kernarbeitsschritte, die aus der Geschäftsprozessmodellierung, Anforderungsanalyse, der objektorientierten Analyse und dem Design, der Implementierung der Anwendung mit darauf folgenden Test sowie der letztlichen Auslieferung bestehen. Die dargestellten Flächen zeigen den Aufwand, der in der jeweiligen Phase in die Kernarbeitsschritte gesteckt werden muss. So erfolgt ein Großteil der Geschäftsprozessmodellierung beim Projektstart und in der Entwurfsphase, die Implementierung findet hauptsächlich beim Entwurf und bei der Konstruktion statt.

Neben der Tatsache, dass ein Kernarbeitsschritt im RUP nie abrupt endet, beispielsweise durch eine Abnahme, lässt sich noch eine zweite Aussage treffen. Vergleichen Sie bitte den Aufwand der Implementierung mit der Summe der restlichen Flächen: Hier ist festzustellen, dass die Implementierung nicht mehr als 20 % des Gesamtaufwands ausmacht, während das Coding beim Wasserfallmodell noch 80 % der Ressourcen in Anspruch genommen hat. Der Aufwand hat sich also von der (objektorientierten) Programmierung (OOP) hin zu der Geschäftsprozessmodellierung, der Anforderungsanalyse, der fachlichen objektorientierten Analyse (OOA) sowie hin zu dem formalen technischen Design der Anwendung (OOD) verlagert.

Von Bedeutung ist weiterhin, dass die Phase des Testens nicht hinter die Implementierung fällt, sondern zum Großteil bereits während der Startphase angesiedelt ist. Wie

kann etwas getestet werden, was noch gar nicht implementiert wurde? Die Antwort liegt in der Spezifikation der Testfälle! Bereits nachdem die ersten Anforderungen an die Anwendung festgeschrieben sind, sollte man festhalten, wie die Umsetzung der Anforderungen getestet werden kann. Dabei sind die Testfälle bereits möglichst frühzeitig festzuhalten.

Neben den grundlegenden Schritten der Entwicklung definiert der RUP weitere unterstützende Arbeitsschritte, die sich durch das gesamte Projekt ziehen. Dazu gehört das Konfigurations- und Änderungsmanagement zur Dokumentation der Anforderungen, deren Erfüllung und deren Änderungen während des Projektverlaufs sowie das Projektmanagement zur Führung, Koordination, Steuerung und Kontrolle der Ressourcen des Softwareentwicklungsprojekts. Die Aktivitäten zur Errichtung der notwendigen Infrastruktur für die zu erstellende Anwendung werden in einem separaten Arbeitsschritt zusammengefasst, der sich ebenfalls über das gesamte Projekt erstreckt.

Die technologische Infrastruktur ist eine Umgebung, in der das Gesamtprodukt entwickelt, zusammengestellt und den Stakeholdern zur Verfügung gestellt wird. Dazu werden einerseits die benötigten Tools der Entwickler und andererseits ein Arbeitsbereich für die Integration aller Teilprodukte zum Gesamtprodukt eingerichtet. Diese Integration wird als Continous Integration bezeichnet und beschreibt den Prozess des regelmäßigen, vollständigen Neubildens und Testens der zu erstellenden Anwendung. Dies geht heutzutage meist einher mit einer Versionsverwaltung der Softwaremodule sowie der Bildung von Revisionen von lauffähigen Prototypen der Anwendung.

Abschließend ist beim RUP-Modell zu betonen, dass jede Phase in mehrere Iterationen unterteilt werden kann, zu denen jeweils Prototypen der Anwendung bzw. der Spezifikationen (OOA/D) erstellt werden. Damit unterstützt RUP die Idee der iterativ/inkrementellen Softwareentwicklung, die heutzutage insbesondere für große Projekte anerkannt ist.

Dies bedeutet, dass die Entwicklung einen Prozess der kontinuierlichen Verbesserung durchläuft, der in kleinen Schritten und mit mehreren Wiederholungen vollzogen wird. Somit werden die Analyse, das Design und der Entwurf mehrfach überarbeitet und können vom Kunden in seine gewünschte Richtung gelenkt werden. Diese an das Spiralmodell angelehnte Vorgehensweise hat sich in der Praxis als üblich herausgestellt, erschwert jedoch eine exakte Zeit- und Kostenplanung im Vorfeld. Man ist in der Softwareentwicklung zu dem Schluss gekommen, dass gerade große Projekte nicht vollständig im Vorfeld „am Reißbrett“ geplant werden können. Lediglich einzelne Meilensteine zur Orientierung können in den frühen Projektphasen definiert werden.

3.2.2 Begriffe der Objektorientierung

Vom Beginn dieses Kapitels bis zu diesem Punkt wurde die historische Vorgehensweise aus Sicht der Softwaretechnik vom Wasserfallmodell bis zum Rational Unified Process beschrieben. Es wurde bereits gesagt, dass die Objektorientierung mit einer iterativ-inkrementellen Vorgehensweise der aktuelle Stand der Technik insbesondere bei großen Projekten und Verwaltungssystemen ist. Die Vorgehensweise besteht aus

- einer Geschäftsprozessanalyse und -modellierung (GPA und GPM)
- einer objektorientierten fachlichen Modellierung (OOA)
- einer objektorientierten technischen Modellierung (OOD)
- einer Umsetzung in (PHP-)Quellcode (OOP)

Ebenso wurde erwähnt, dass die Objektorientierung ein Ansatz ist, der sich näher an der benötigten Funktionalität befindet, die der Kunde wünscht und weniger nah an technische Details. Die Objektorientierung verlangt also eine eigene Denkweise und besitzt auch einen eigenen Wortschatz, der für jede objektorientierte Programmiersprache identisch ist. Dieser Wortschatz und der Ansatz der Objektorientierung wurde bislang jedoch noch nicht vorgestellt. Dies geschieht in diesem Kapitel.

Objekt und Klasse

Von zentraler Bedeutung der Objektorientierung sind die Begriffe „Objekt“ und „Klasse“. Ein Objekt ist ein Element der realen Welt, das in der zu erstellenden Anwendung abgebildet bzw. repräsentiert werden soll. Dabei kann es sich um einen materiellen Gegenstand, ein Lebewesen, aber auch um einen Vorgang oder um eine betriebliche Organisationseinheit, beispielsweise um die Verkaufsabteilung oder um einen konkreten Tarif TvöD 12 handeln. Jedes Objekt hat einen inneren Zustand, ein Verhalten zur Umwelt und eine Identität, die das Objekt eindeutig identifiziert und es somit von anderen Objekten unterscheidet.

Aus Sicht eines Entwicklers ist ein Objekt mehr als nur eine Zahl, ein Wert oder eine Zeichenkette. Es ist stets aus verschiedenen Elementen zusammengesetzt.

Beispiel

Herr Meier hat den Beruf eines Verkäufers in dem Autohaus EuroCar. Frau Schulz betritt das Autohaus und möchte sich erst einmal umsehen. Herr Meier sieht das Kaufinteresse und spricht Frau Schulz an. Sie interessiert sich insbesondere für Familienwagen, da sie verheiratet ist und 2 Kinder hat. Außerdem bevorzugt sie rote Autos; sie findet die Farbe schön. Herr Meier findet im Gespräch heraus, dass Frau Schulz gern an Wochenenden Familienausflüge im Umkreis von 500km von ihrem Wohnort durchführen will. Herr Meier berät sie zuerst auf den sportlichen Kombi Y3 mit 250PS. Er hat genügend Leistung für Autobahnfahrten und kostet 53 000 Euro. Frau Schulz lehnt nach 10 Minuten dankend ab. Nun versucht Herr Meier sein Glück bei dem neuen i2000, der an einen kleinen Bus erinnert. Dieses Auto hat 140 PS und einen Hybridantrieb, der 70 % weniger Benzin verbraucht als der Y3. Von der geräumigen Ausstattung ist Frau Schulz direkt überzeugt. Die beiden gehen in das Büro von Herrn Meier und schließen einen Kaufvertrag ab. Frau Schulz möchte in einer Woche die 42 000 Euro bar bezahlen und das Auto aus der Ausstellung dann auch sofort mitnehmen. Zusätzliche Ausstattung wünscht sie nicht.

Der Besitzer des Autohauses dieses ersten Beispiels hat Ihnen das oben beschriebene realitätsnahe Szenario mündlich geschildert. Er hat den Wunsch, dieses gesamte Szenario in einer Software zu protokollieren, die Sie erstellen sollen. Herr Meier soll, sobald Frau

Schulz das Autohaus verlassen hat, den gesamten Ablauf über eine PHP-Anwendung dokumentieren. Sein PC hat einen entsprechenden Internetbrowser installiert.

Profitipp

Versuchen Sie als Entwickler bzw. als Analytiker, Ihrem Kunden solche Szenarien zu entlocken und protokollieren Sie diese. Die geschilderten Abläufe bieten einen idealen Einstieg in die Fachdomäne Ihres zukünftigen Kunden.

Nach diesem ersten Ausschnitt aus der Realität ist es nun Ihre Aufgabe, die vorhandenen Objekte zu identifizieren. Eine sinnvolle Identifikation ist maßgeblich für eine gute Modellierung, jedoch kann kein Algorithmus angegeben werden, der ein Objekt identifiziert. Es gibt lediglich Methoden wie die Verb-Substantiv-Analyse oder die Methode der CRC-Karten, die im weiteren Verlauf dieses Kapitels erläutert werden. Diese Methoden sollen bei der Ermittlung der Objekte und Klassen helfen. Im Wesentlichen ist jedoch Ihr Geschick bzw. Gefühl als Systemanalytiker gefragt. Aus dem Text des Beispiels können die folgenden Objekte identifiziert werden:

- Herr Meier
- Autohaus EuroCar
- Frau Schulz
- Kombi Y3
- Bus i2000
  - Hybridantrieb
  - Ausstattung
- Büro von Herrn Meier
- Kaufvertrag zwischen Herrn Meier und Frau Schulz

Dabei handelt es sich ausschließlich um materielle Objekte. Ob Sie alle Objekte in Ihrer Anwendung abbilden oder nicht, wird später zusammen mit dem Kunden entschieden. Dann ist die Frage zu stellen, welchen Beitrag Ihre Software leisten soll. Meist schwieriger sind für Anfänger Objekte zu identifizieren, die man nicht greifen kann. Solche Objekte sind in dem Beispiel:

- Verkaufsgespräch zwischen Herrn Meier und Frau Schulz
- Kaufinteresse von Frau Schulz

Welchen Sinn macht es, ein Verkaufsgespräch und ein Kaufinteresse in einer Software abzubilden? Für eine Geschäftsleitung können solche Daten sehr bedeutend sein, beispielsweise, um das Kaufverhalten potenzieller Kunden oder Interessen und Trends zu analysieren. Falls es nicht zum Kauf kommt, stellt sich die Frage, was der Verkäufer beim nächsten Mal besser machen kann. Vielleicht ist auch die Produktpalette zu verbessern, wenn der Trend hin zu sparsamen Familienwagen geht. Solche Analysen will der Inhaber des Autohauses in diesem Fall wahrscheinlich durchführen. In einem nächsten Gespräch sollten Sie ihn also nach dem Zweck der zu erstellenden Anwendung fragen.

Nachdem Sie aus realen betrieblichen Abläufen Objekte identifiziert haben, müssen Sie diese zu Klassen gruppieren. Es erfolgt also eine weitere Abstraktion, noch bevor Sie mit der Programmierung der ersten Zeile Quellcode beginnen. Eine Klasse ist im nächsten Schritt ein Bauplan, um gleichartige Objekte zu erzeugen; sie beschreibt also Objekte. Lassen Sie sich auch hierbei von der Realität leiten, denn im Alltag klassifizieren Sie bereits sehr oft.

Wenn Sie beispielsweise von einer Brücke auf eine Autobahn schauen, so fahren dort *Fahrzeuge*, nämlich *Autos* und einige *LKWs*. Sie beginnen nicht, die Fahrgestellnummern der einzelnen konkreten Gefährte aufzuzählen. Bei einer Klassifizierung betrachten Sie also Mengen bzw. Sammlungen von Objekten. Sie wissen auch, dass *Fahrzeuge* sowohl *LKWs* als auch *Autos* bzw. *PKWs* beinhalten. Sie können demnach mehrfache Klassifizierungen durchführen. Abbildung 3.12 skizziert die Abstraktion der Realität zu einem Objekt und von einem Objekt zu einer Klasse.

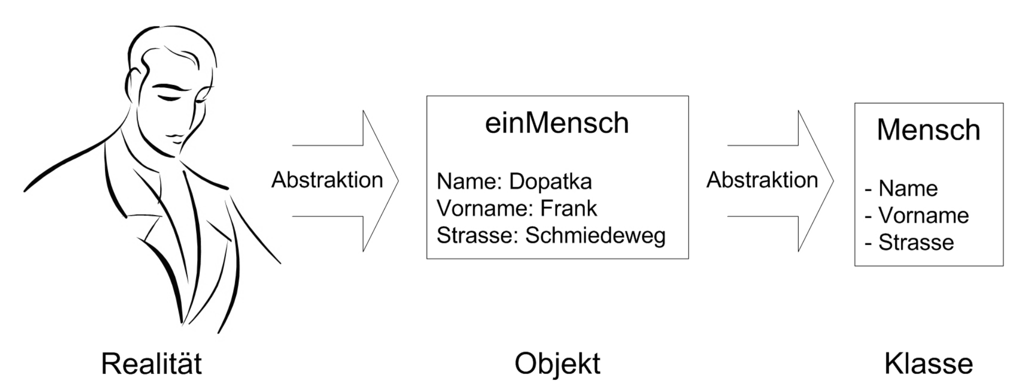


Abbildung 3.12: Abstraktion der Realität zu einer Klasse

Ein konkretes Objekt ist eine Instanz, also ein existierendes Exemplar einer Klasse. Das Objekt wiederum ist ein Modell eines Ausschnitts aus der Wirklichkeit. Je präziser die Modellierung erfolgt, desto besser kann die Wirklichkeit später in der objektorientierten Anwendung abgebildet werden. Wie kann im Beispiel des Autohauses eine Abbildung der Objekte auf Klassen aussehen?

- Herr Meier -> Verkäufer -> Mitarbeiter -> Person
- Autohaus EuroCar -> Autohaus
- Frau Schulz -> Kunde -> Person
- Kombi Y3 -> PKW -> Fahrzeug -> Artikel
- Bus i2000 -> PKW -> Fahrzeug -> Artikel
  - Hybrid-Antrieb -> Antrieb -> Ausstattung
  - konkrete Ausstattung -> Ausstattung
- Büro von Herrn Meier -> Raum
- Kaufvertrag zwischen Herrn Meier und Frau Schulz -> Kaufvertrag -> Vertrag

Auch immaterielle Objekte müssen zu Klassen zusammengefasst werden:

- Verkaufsgespräch zwischen Herrn Meier und Frau Schulz
  -> Verkaufsgespräch -> Gesprächsprotokoll
- Kaufinteresse von Frau Schulz -> Kaufinteresse

Treten Personen wie Herr Meier oder Frau Schulz als Objekte auf, so schlüpfen sie meist in so genannte *Rollen*, die im System von Interesse sind. Herr Meier übernimmt die Rolle des Verkäufers und Frau Schulz die eines Kunden. Diese Rollen bilden dann die Klassen.

EuroCar ist ein konkretes Autohaus, bei dem Herr Meier angestellt ist. Die zu erstellende Anwendung soll jedoch auch auf andere Autohäuser übertragen werden können und ggf. sogar mehrere Autohäuser verwalten. Somit existiert auch eine Klasse *Autohaus*.

Der vorgestellte Kombi und der Bus sind beide PKWs. Ein PKW ist ein Fahrzeug. Da in einem Autohaus Fahrzeuge verkauft werden, werden die beiden PKWs als Artikel gehandhabt, die man kaufen kann. Bei dem Bus i2000 wurde erwähnt, dass der Antrieb und die Ausstattung von Bedeutung beim Verkaufsvorgang sind. Da es verschiedene Antriebe und Ausstattungsmerkmale gibt, können auch diese zu Klassen zusammengefasst werden.

Das Büro von Herrn Meier ist ein Raum. Zusätzlich existieren u. a. noch der Verkaufsraum und das Büro vom Chef. Dass die Anwendung später auch Räume verwalten soll, ist eher unwahrscheinlich.

Auch das Vertragsobjekt kann ebenso wie das durchgeführte Verkaufsgespräch jeweils als Klasse hinterlegt und damit von der Anwendung verwaltet werden. Das Verkaufsgespräch kann man als spezielles Gesprächsprotokoll sehen. Ein anderes Gesprächsprotokoll wäre beispielsweise das Protokoll bei einem Meeting zwischen dem Chef und seinen Angestellten. Es kann sogar Sinn machen, das Kaufinteresse aller Kunden zu klassifizieren, um damit Prognosen für Trends und Statistiken im System durchzuführen.

Ein zweites Beispiel – diesmal aus der Architektur – soll den Unterschied zwischen einer Klasse und konkreten Objekten verdeutlichen. Bevor Sie bauen, also konkrete Objekte erstellen, müssen Sie zunächst einen Bauplan bei einem Architekten erstellen. Dieser Bauplan für ein Haus entspricht einer Klasse in der Softwareentwicklung.

Anhand dieses einen Bauplans können Sie beispielweise fünf konkrete Häuser nebeneinander bauen lassen. Dies sind die Objekte, die aus der Spezifikation der Klasse entstanden sind. Obwohl sich die Farbe, die Fenster (Holz oder Kunststoff), die Form der Haustür und die Gestaltung der Inneneinrichtung bei den einzelnen Objekten unterscheiden, sieht man den Häusern an, dass sie nach einem einzigen Plan gebaut wurden. Den Effekt kann man besonders gut bei Reihenhaussiedlungen erkennen.

In einem dritten Beispiel sollen die Klassen für eine Aktienverwaltung ermittelt werden. Mit der Anwendung soll man sein Depot – also seinen Aktienbestand – als PHP-Anwendung verwalten können. Sowohl die Aktienverwaltung als auch das Depot sind dabei Klassen in der zukünftigen objektorientierten Anwendung. In einer Aktienverwaltung kann man unter Umständen mehrere Depots verwalten. Ein Depot beinhaltet verschiedene Aktien. Jede Aktie hat einen Kurs, wobei sich die Kurse an verschiedenen Handelsplätzen leicht unterscheiden können. Ein Aktienkurs ist also einem Handelsplatz zugeordnet. Will man Aktien kaufen oder verkaufen, so geschieht dies nicht unmittelbar. Sie

können in der Regel nicht direkt einen Kauf bzw. Verkauf durchführen. Stattdessen setzen Sie eine Order ab. Mit einer Verkaufsorder bieten Sie eine Menge von Aktien eines Typs zum Verkauf an einem bestimmten Handelsplatz an; mit einer Kauforder signalisieren Sie, dass Sie gern Aktien kaufen wollen. Wenn ein Partner am Handelsplatz mit Ihrer Order einverstanden ist, werden der Kauf bzw. der Verkauf durchgeführt. Aus dieser Beschreibung lassen sich folgende Klassen erkennen:

- Aktienverwaltung
- Depot
- Aktie
- Kurs
- Handelsplatz
- Order, entweder Kauforder oder Verkaufsorder

Was es hat und was es kann: Eigenschaften und Methoden

Wie geht es nun nach der Identifikation der Klassen weiter? Aus was besteht eine Klasse? Ein solcher Bauplan für die Erstellung von Objekten besteht aus zwei Teilen, den Eigenschaften und den Methoden.

Eigenschaften werden auch als Attribute der Klasse bezeichnet, die jedes erzeugte Objekt aus dieser Klasse kennzeichnen. Programmierer aus prozeduralen Sprachen nennen die Eigenschaften auch Variablen oder Daten. Sie besitzen jeweils einen Datentyp aus der verwendeten Programmiersprache.

Jeder Stift verfügt beispielsweise über die Eigenschaft, dass er eine Farbe und einen Füllstand besitzt. Eine Person hat einen Namen und einen Vornamen. Wenn ein Kunde eine Person ist, dann besitzt dieser auch einen Namen, einen Vornamen und zusätzliche Eigenschaften. Eine Aktie hat einen Namen, eine ISIN (International Securities Identification Number), einen aktuellen Kurs an jedem Handelsplatz. Jeder Kurs besteht aus seinem Handelsplatz, einem Währungswert und einem Datums- und Zeitwert. Wenn Sie nach den Eigenschaften einer Klasse suchen, müssen Sie sich die Frage stellen:

- Was hat jedes Objekt dieser Klasse?
- Wie kann man es beschreiben?
- Aus welchen Teilen besteht es?

Aus dem Beispiel des Autohauses wurden die folgenden Hauptklassen identifiziert:

- Autohaus
- Person
- Artikel
- Antrieb
- Ausstattung
- Vertrag
- Gesprächsprotokoll
- Kaufinteresse

Wenn Sie sich nun die oben genannten Fragen stellen, können Sie zu dem in Abbildung 3.13 dargestellten Ergebnis kommen. Wichtig ist auch zu unterscheiden, ob die Eigenschaften, die Sie finden, für wirklich jedes der Objekte zutreffen oder optional sind. Eine Person, die Sie in Ihrer Anwendung beschreiben wollen, hat beispielsweise auf jeden Fall einen Namen. Aber nicht jede Person muss über einen Führerschein verfügen, um ein Auto kaufen zu können.

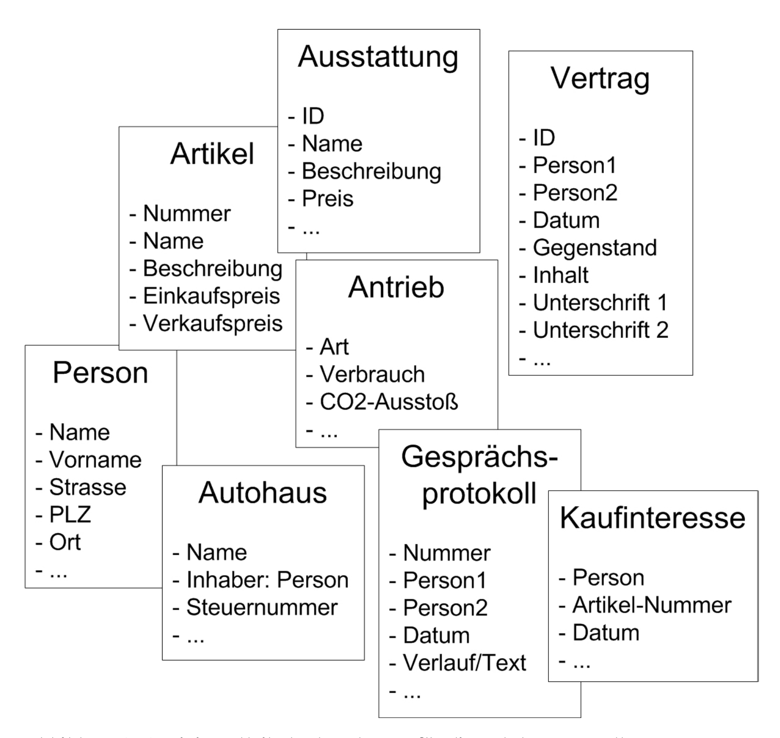


Abbildung 3.13: Einige Attribute der Klassen für die Autohausverwaltung

Wie Sie sehen, muss nicht jede Eigenschaft ein elementarer Wert wie eine Zahl oder eine Zeichenkette sein. Sie kann auch ein Verweis auf ein anderes Objekt darstellen. Der Inhaber eines Autohauses ist beispielsweise eine Person.

Sie können die Erstellung der Klassen mit ihren Eigenschaften gut vergleichen mit der ER-Modellierung einer Datenbank. Dort existiert eine Tabelle *Kunde* mit den Feldern *Name*, *Vorname*, *Strasse*, *PLZ*, *Ort* usw. Die Spezifikation der Tabelle in der Datenbank mit dem SQL-Befehl *CREATE TABLE Kunde ...* ist vergleichbar mit der Beschreibung der Klasse mit ihren Eigenschaften *Name*, *Vorname*, *Strasse*, *PLZ*, *Ort* usw. Ein konkreter Kunde mit dem Namen *Müller*, dem Vornamen *Uli* usw. stellt genau einen Datensatz in der Datenbanktabelle dar. Diese Daten sind konkrete Wertausprägungen der Eigenschaften.

Während die Eigenschaften nur die Daten eines Objekts beschreiben, legen die Methoden dessen Fähigkeiten fest. Eine Methode kann in der Objektorientierung auch als Operation bezeichnet werden. In der prozeduralen Programmierung wurde eine Methode als Funktion, Prozedur oder Unterprogrammaufruf bezeichnet. Um eine Methode mit Funktionalität zu füllen, verwenden Sie bereits bekannte Programmier-Techniken wie

sequentielle Anweisungen, Verzweigungen und/oder Schleifen. Außerdem können in Methoden andere Objekte erstellt, angesprochen und verwaltet werden.

SQL legt DB-Tabelle an...

```
CREATE TABLE Kunde (
ID INTEGER NOT NULL PRIMARY KEY,
Name VARCHAR(50) NOT NULL,
Vorname VARCHAR(50) NOT NULL,
Strasse  VARCHAR(50) NOT NULL,
PLZ  VARCHAR(10) NOT NULL,
Ort VARCHAR(50) NOT NULL);
```

Klassenbeschreibung in PHP ...

```
class Kunde{
  private $ID;
  private $Name;
  private $Vorname;
  private $Strasse;
  private $PLZ;
  private $Ort;
}
```

Ein konkreter Tabelleneintrag...

ID	Name	Vorname	Strasse
342	Müller	Uli	Hauptstrasse ...

Ein konkretes Objekt...



Abbildung 3.14: Klassen und Eigenschaften vs. Datenbankmodellierung

Um an die Methoden der bereits identifizierten Klassen zu kommen, wenden Sie am besten folgende Fragestellungen an:

- Was kann man damit machen?
- Über welche Funktionen verfügt es?

Die Antworten sollten stets aus Verben bestehen. So kann man beispielsweise mit einem Stift *schreiben*. Außerdem kann man einen Stift *öffnen*, *schließen* und *nachfüllen*. Dies alles sind Methoden/Funktionen, die man mit einem Stift ausführen kann. Abbildung 3.15 skizziert zwei Stiftobjekte mit ihren aktuellen Eigenschaftswerten und ihren Methoden. Man muss jedoch bedenken, dass man nicht jeden Stift öffnen, schließen und nachfüllen kann. Ein Buntstift bietet diese Funktionen beispielsweise nicht. Die Abbildung zeigt also bereits sehr spezielle Stifte. Rechts in der Abbildung ist die Klassifizierung der einzelnen Objekte zu der Klasse *Stift* dargestellt. Die Namen der Attribute, deren Datentypen sowie die anwendbaren Methoden gelten also für jeden Stift und sind Teil der Beschreibung des Bauplans, also der Klasse. Die Belegung der Eigenschaften, also deren konkrete Wertausprägung, gehört hingegen zu jedem Objekt. Die Summe der gesetzten Werte bildet den inneren Zustand des existierenden Objekts bzw. Exemplars einer Klasse.

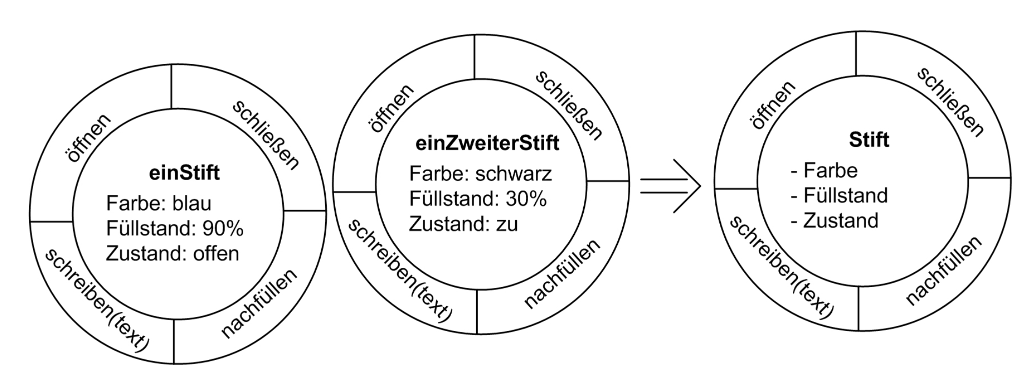


Abbildung 3.15: Zwei Stiftobjekte und deren Klassifizierung

Für das Beispiel des Autohauses bedeutet dies, dass Sie die Funktionen der 8 Hauptklassen ermitteln müssen. So können Sie beispielsweise einen Artikel anlegen, ändern, aus dem Bestand entfernen, nach einem Artikel suchen, seine Daten ausgeben, seinen Einkaufspreis neu setzen usw. Auch eine Person können Sie im System anlegen, ihre Daten aktualisieren und ausgeben, nach ihr suchen oder sie löschen.

Im Beispiel der Aktienverwaltung können Sie beispielsweise eine Kauforder platzieren mit oder ohne Limit oder den Status bzw. den Zustand der Order einsehen wie *ausgeführt*, *teilweise ausgeführt*, *in Bearbeitung*, *abgelaufen* oder *storniert*. Außerdem können Sie eine noch nicht vollständig ausgeführte Order noch ändern, sie stornieren oder sich einfach die Daten der Order ansehen. Dies sind alles Methoden der Klasse *Order*.

Um festzustellen, welche Methoden alle existieren und wie sie intern ablaufen, benötigen Sie die entsprechende Kenntnis aus der Fachdomäne. Die benötigte Funktionalität erfahren Sie am besten von den zukünftigen Anwendern des Systems. Wenn Sie in der Rolle des Systemanalytikers sind, müssen Sie sich also eine gewisse Kenntnis über die Fachdomäne aneignen.

Öffentlich, privat und etwas dazwischen: Sichtbarkeiten

Eine Neuerung in der Objektorientierung gegenüber der prozeduralen Programmierung liegt in der strengen Datenkapselung. So soll auch ein anderer Entwickler nie direkt von außen auf eine Eigenschaft eines Objekts zugreifen können. Der Zugriff soll ausschließlich über einen Methodenaufruf erfolgen. Derjenige, der ein Objekt verwaltet, muss also eine Botschaft bzw. eine Nachricht an das Objekt schicken mit der Bitte, eine Eigenschaft neu zu belegen. Ob und wie das Objekt dieser Bitte nachkommt, ist einzig und allein Angelegenheit des Objekts selbst. Die Hoheit zur Änderung des inneren Zustands liegt also beim Objekt.

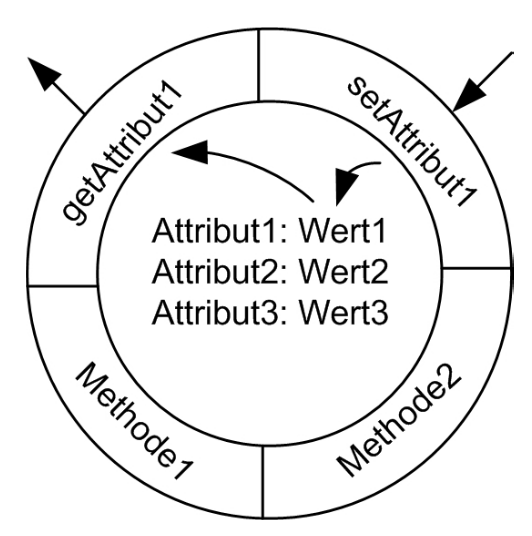


Abbildung 3.16: Zugriff auf eine Eigenschaft/Attribut über Methodenaufrufe

Um eine Eigenschaft zu schreiben, muss man eine so genannte Set-Methode aufrufen. Der Quelltext der Set-Methode entscheidet dann, ob und wie die Eigenschaft geändert wird. Um eine Eigenschaft auszulesen, verwendet der Verwalter des Objekts eine entsprechende Get-Methode, die den Wert der Eigenschaft zurückliefert. Wenn eine Eigenschaft von außen nicht geändert werden kann, existiert keine Set-Methode. Ist eine interne Eigenschaft nicht auslesbar, so existiert keine Get-Methode. Andere Methoden wie *ändernDaten* der Klasse *Artikel* können komplexere Änderungen der internen Eigenschaften vornehmen. Generell gelten folgende Regeln:

1. Eigenschaften sind *private*, also nicht von außen zugreifbar, zu deklarieren. Sie werden vom Objekt selbst verwaltet und müssen geschützt werden.
2. Methoden sind die Dienste des Objekts und sind daher als *public* zu deklarieren. Ausnahmen bilden nur Hilfsfunktionen zur internen Berechnung, die nach außen nicht sichtbar sein sollen.

Als Beispiel kann die Farbe eines Stifts herhalten, die nachträglich geändert wird, indem beispielsweise ein Kugelschreiber eine neue Mine erhält. Ein dummer Ansatz besteht in der öffentlichen Definition einer Eigenschaft *Farbe* als Zeichenkette. Denn so könnte man dem Stift mit dem Aufruf *einStift.farbe="Frank"* eine sinnlose Farbe zuweisen. Damit der Stift diese Zuweisung prüfen kann, muss die Farbe *private* deklariert sein und der Stift eine Methode *setFarbe* besitzen, die eine Zeichenkette als Parameter bekommt und als *public* deklariert ist. Der Aufruf würde dann mit *einStift->setFarbe("Blau")* erfolgen. Da eine Farbe selbst in Wirklichkeit keine Zeichenkette, sondern ihrerseits wieder eine Zusammensetzung von Eigenschaften ist, lohnt es sich, auch die Klasse *Farbe* einzuführen. Ein solches Objekt kann beispielsweise je 0 bis 255 Rot-, Grün- und Blau-Anteile besitzen. Dann wäre es ein RGB-Farbobjekt, das dem Stift in der Methode *setFarbe* übergeben werden könnte.

Neben den Sichtbarkeiten *public* und *private* existiert noch eine dritte Sichtbarkeit, die als *protected* bezeichnet wird. Eine Eigenschaft oder eine Methode mit *protected*-Deklaration

verhält sich dabei wie *private* für fremde Klassen und wie *public* für vererbte Klassen. Man erlaubt also seiner eigenen Kindklasse einen tieferen Eingriff in die Privatsphäre als einer fremden Klasse. Wie die Vererbung prinzipiell funktioniert und welche Bedeutung sie hat, wird im nächsten Unterkapitel erläutert.

Spezialisieren und Generalisieren: Vererbung

Bereits bei dem Ermitteln der Klassen im Beispiel des Autohauses ist eine Kette von Klassen ermittelt worden, nämlich zwischen Verkäufer, Mitarbeiter und Person sowie zwischen Käufer und Person. Klassen können miteinander über eine Vererbung verkettet sein. In der Analyse erkennen Sie diese Ketten stets durch die „Ist ein“-Beziehung. So ist der Verkäufer ein Mitarbeiter des Autohauses und jeder Mitarbeiter des Autohauses ist eine Person.

Sowohl Mitarbeiter als auch Kunden und Lieferanten sind Personen. Genauso sind PKWs, LKWs und Züge Fahrzeuge. Man spricht in dieser Richtung von einer Generalisierung. Der Kombi Y3 ist ein spezieller PKW und ein PKW ist ein spezielles Fahrzeug. Durchläuft man die Klassenhierarchie also in die andere Richtung, so spricht man von einer Spezialisierung.

Es stellt sich die Frage, warum das Finden von Klassenhierarchien von so großer Bedeutung in der Objektorientierung ist und welchen Nutzen man von einer Klassenhierarchie hat. Die Antwort liegt im Wesentlichen in der Vermeidung von doppeltem Quellcode durch die logische Unterteilung. Da sowohl der Kunde als auch der Mitarbeiter Personen sind, müssen die Eigenschaften und auch die Methoden einer Person nur einmalig kodiert werden. Die Klasse *Kunde* und *Mitarbeiter* erbt dann alle Eigenschaften und Methoden von der Oberklasse.

Weder vererbte Eigenschaften noch vererbte Methoden können in den Unterklassen gelöscht oder verworfen werden. Man kann also sein Erbe nicht leugnen. Dies bedeutet, dass man sehr sorgfältig prüfen muss, ob die Eigenschaften und Methoden der Oberklasse wirklich allgemeingültig sind. Ansonsten müssen sie ggf. in die Unterklassen ausgelagert werden.

Eine sinnvolle Struktur der Objektorientierung besteht darin, vererbte Methoden neu zu definieren. Man spricht hier von einem „Überschreiben“ der Funktionalität.

Nehmen wir an, Sie modellieren eine Klasse *Tier* und möchten, dass jedes Tier einen Laut geben kann. Sie definieren also die Methode *gibLaut*. Ein Hund gibt aber einen anderen Laut als eine Katze, obwohl beides Tiere sind. Daher definieren Sie die Methode *gibLaut* für einen Hund ebenso neu wie für eine Katze. Ein Terrier ist wiederum ein spezieller Hund. Diese Klasse von Hunden kann sich dadurch auszeichnen, dass sie anders bellt als ein gewöhnlicher Hund. Auch hier definieren Sie die Funktionalität neu.

Wenn eine Eigenschaft, wie im vorherigen Kapitel gefordert, als *private* markiert ist, kann es von einem Objekt der Unterklasse nicht zugegriffen werden. Damit ein Objekt der Unterklasse Zugriff erhalten kann, muss es wie jedes fremde Objekt auch die entsprechende Get- bzw. Set-Methode ausführen.

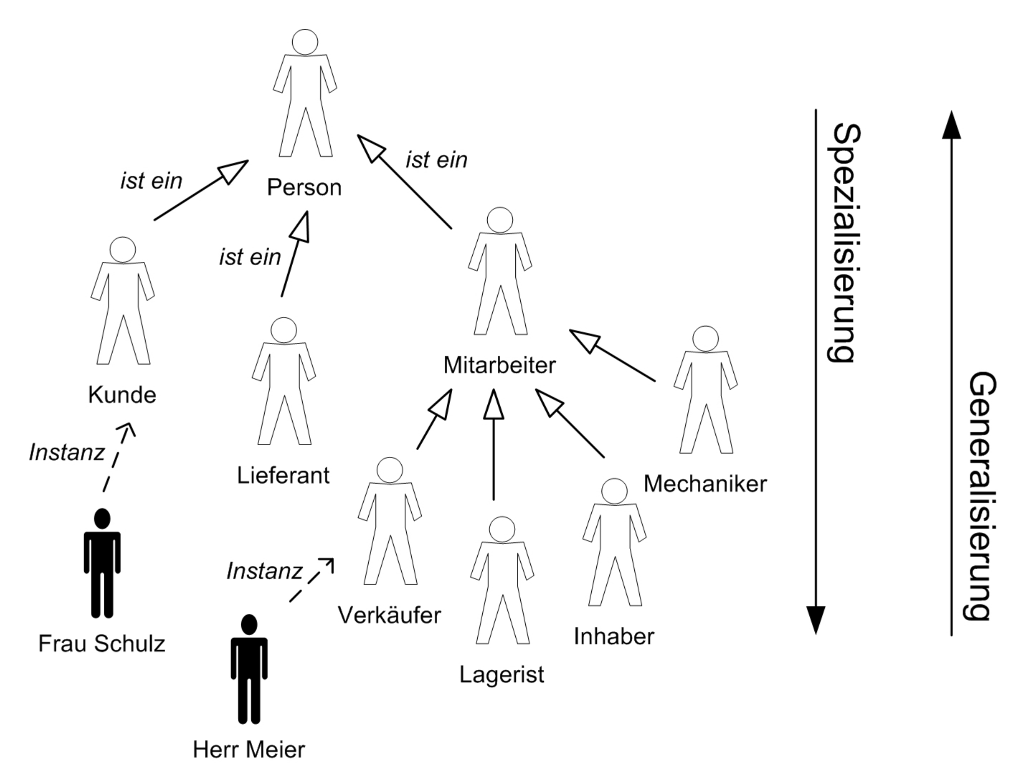


Abbildung 3.17: Vererbungshierarchie von einer Person zu Kunden und Verkäufern

In Abbildung 3.17 kann man beispielsweise spezifizieren, dass jede Person einen Namen hat. Wenn ein anderes Objekt nach dem Vornamen von Herrn Meier fragt, muss dieser wiederum die Methode *getName* vom Verkäufer aufrufen, falls diese Methode überschrieben wurde. So gelangt man irgendwann zur Methode *getName* der Person, die dann wiederum die Zeichenkette ausliest und über die Kette der Methodenaufrufe zurückgibt. Wenn man jeder Person das Recht geben will, direkt auf den Namen zuzugreifen – was ja durchaus sinnvoll ist – so kann man diese Eigenschaft als *protected* deklarieren. Herr Meier als Instanz des Verkäufers kann dann so auf die Eigenschaft zugreifen, als wäre sie direkt in seiner Klasse deklariert worden.

Polymorphie

Ein sehr interessantes Konzept in Verbindung mit Vererbung in objektorientierten Sprachen ist die Polymorphie. Polymorphie beschreibt die Fähigkeit einer Eigenschaft oder Methode, sich abhängig von ihrer Verwendung unterschiedlich darzustellen. Sie erlaubt der Eigenschaft oder Methode, je nach Kontext einen unterschiedlichen Datentypen anzunehmen. Die Bedeutung dieser Definition ist zunächst schwer zu erfassen.

Anhand einiger Beispiele wird das Prinzip der Polymorphie jedoch deutlich: Nehmen wir an, Sie erstellen wie bereits beschrieben eine Klasse *Tier* mit den Unterklassen *Katze* und *Hund*. Sie definieren, dass jedes Tier einen Laut von sich geben muss und erstellen

daher in der Klasse *Tier* die Methode *gibLaut*, die Sie in den Klassen *Katze* und *Hund* überschreiben. Dies ist mit Pseudocode in Abbildung 3.18 skizziert.

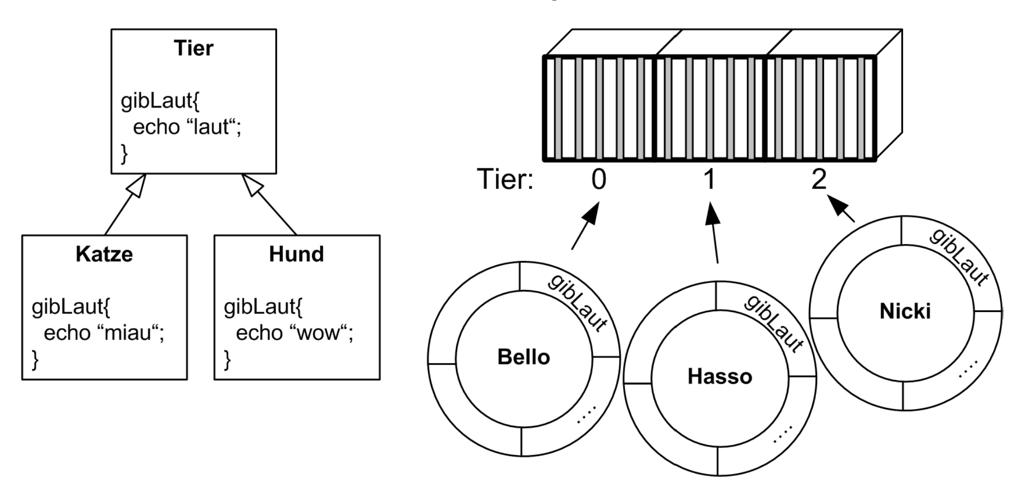


Abbildung 3.18: Polymorphie am Beispiel von Tieren, Hunden und Katzen

Nun erzeugen Sie ein Datenfeld mit 3 Elementen vom Datentyp *Tier*. Das Feld ist in der Abbildung 3.18 als drei Käfige dargestellt, in die beliebig große Tiere hineinpassen. Im Anschluss daran erzeugen Sie zwei Hundobjekte sowie ein Katzenobjekt und weisen diese dem Datenfeld zu. Dies ist möglich, da sowohl Hunde als auch Katzen Tiere sind und diese ja in das Datenfeld passen.

Wenn Sie jetzt an jedem Käfig rütteln, welchen Laut würden Sie hören? Aus technischer Sicht könnten Sie sagen: In jedem Element des Feldes ist ein Tier, also hört man das *gibLaut* der Klasse *Tier*. Dies ist jedoch – wie auch in der Wirklichkeit – falsch. Das Datenfeld ist lediglich ein Speicherbereich, in dem man ein beliebiges Tier unterbringen kann, also eine Reihe von Käfigen. In jedem Käfig befindet sich ein spezielles Tier, entweder eine Katze oder ein Hund. Sobald Sie an dem Käfig rütteln, hören Sie den Laut des Tieres, das sich gerade in dem Käfig befindet. Genauso ist es auch in der Wirklichkeit.

Ein zweites Beispiel: Sei *Form* eine Basisklasse und habe die Methode *zeichnen*. Wenn *Kreis* und *Linie* beide von *Form* abgeleitet sind, müssen sie sinnvollerweise die *zeichnen*-Methode von *Form* überschreiben. Denn jedes Objekt weiß selbst, wie es gezeichnet werden kann. Eine andere Klasse kann nun mit einem Datenfeld von Formen arbeiten, und auf jedem Formobjekt die *zeichnen*-Methode aufrufen. Der in PHP integrierte Polymorphiemechanismus bewirkt, dass immer die Methode des eigentlichen Objekts aufgerufen wird.

Konstruktor und Destruktor

Es wurde bereits gesagt, dass die Klasse eine Beschreibung darstellt, um Objekte zu erzeugen. Ein Objekt hat dann einen inneren Zustand und Methoden, über die es angesprochen werden kann. In jeder objektorientierten Programmiersprache existiert nun ein Befehl, um ein neues Objekt zu erzeugen. Dieser Befehl heißt in PHP wie auch in Java *new*. Als Rückgabe des Befehls erhält man eine Referenz, also ein Zeiger auf das gerade

erstellte Objekt. Es ist Pflicht in der Objektorientierung, dass jedes Objekt zu jedem Zeitpunkt, also auch direkt nach seiner Erzeugung, in einem gültigen Zustand ist.

Um diesen gültigen Zustand zu erreichen, können Sie Vorschriften zur Erzeugung von neuen Objekten Ihrer eigenen Klasse definieren. Diese Vorschriften nennen sich Konstruktoren. Ein Konstruktor kann eine Liste von Eingabeparametern erhalten, die für die Erstellung eines Objekts dieser Klasse zwingend notwendig sind. Gegebenenfalls existieren mehrere Möglichkeiten zur Erzeugung eines Objekts. Daher können auch mehrere Konstruktoren existieren.

Ein Beispiel ist die Klasse *Bruch* mit den Eigenschaften *Zähler* und *Nenner*. Wenn Sie einfach einen Bruch erzeugen, ist sowohl der Zähler als auch der Nenner 0. Dies ist jedoch laut Definition kein gültiger Bruch. Sie müssen also zumindest einen Konstruktor definieren, der den Nenner auf einen Wert ungleich 0 setzt, um einen gültigen Bruch zu erzeugen.

Ein anderes Beispiel ist die Erzeugung eines Kunden im System. Es macht keinen Sinn, einen Kunden zu erzeugen, der keinen Namen und/oder keine Anschrift hat. Stellen Sie sich vor, Sie würden diesem existierenden Kunden im Anschluss an dessen Erzeugung eine Rechnung zuweisen. Sie müssen sich also Gedanken machen, welche Eigenschaften zwingen gesetzt werden müssen. Das Beispiel des Kunden können Sie wieder mit der Modellierung einer Datenbanktabelle vergleichen. Dort haben Sie bei der Erstellung einer Tabelle die Möglichkeit, Feldern einen Initialwert zuzuweisen oder zumindest zu sagen, dass eine Spalte nicht *NULL* enthalten darf. Mit einem Konstruktor haben Sie noch weitaus mehr Möglichkeiten, da sie den gesamten Funktionsumfang von PHP nutzen können. Sie können beispielsweise innerhalb eines Konstruktors weitere Objekte anlegen, die für die Existenz des eigenen Objekts unabdingbar sind.

Wie bereits erwähnt wurde, kann ein Objekt nicht nur elementare Datentypen wie Zeichenketten oder Zahlen als Eigenschaften haben, sondern auch andere Objekte. So besteht der Bus i2000 des Autohausbeispiels aus einem Hybridantrieb und einer speziellen Ausstattung. Wenn der konkrete Bus i2000 aus dem Bestand des Autohauses entfernt werden sollte, so können auch die Daten zu seinem Antrieb und seiner Ausstattung entfernt werden.

Ein ähnlicher Fall ergibt sich, wenn Sie ein Datenverbindungsobjekt zu einer Datenbank oder zu einer Datei verwalten. Wenn das Objekt entfernt wird, sollte man noch die geöffnete Verbindung zum Datenbankserver schließen bzw. die geöffnete Datei schließen.

Für diese Fälle bietet PHP eine spezielle Destruktormethode. Sie wird aufgerufen, sobald das Objekt gelöscht wird. Der Destruktor dient dazu, auch nach dem Entfernen des Objekts einen konsistenten Zustand im Speicher des Webservers zu hinterlassen.

Sich kennen lernen: Assoziationen

Wenn ein Objekt A eine Referenz auf ein anderes Objekt B besitzt, so kann das Objekt A auf alle Methoden von Objekt B zugreifen. Man sagt, Objekt A kennt Objekt B. Wenn sich Objekte von zwei Klassen kenne können, besitzen sie eine Assoziation zueinander. Abbildung 3.19 zeigt, dass die Assoziationen sehr komplexe Geflechte von Objekten bilden können, die die Wirklichkeit modellieren.

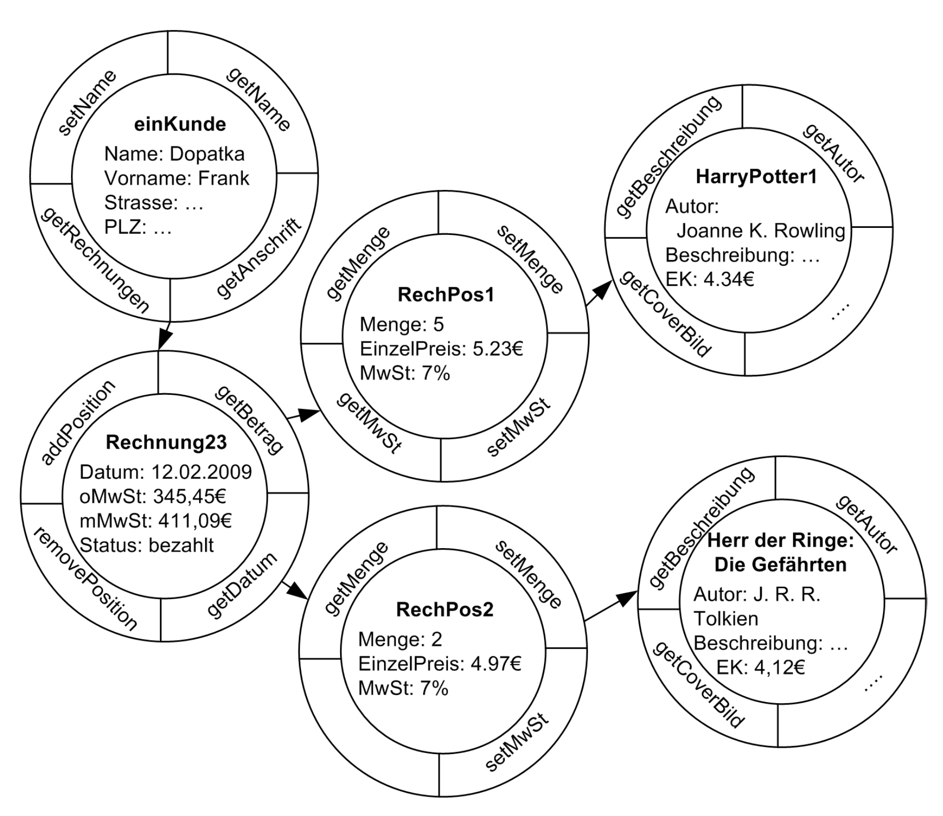


Abbildung 3.19: Assoziationen zwischen Objekten

Die Abbildung zeigt, dass ein Kunde eine Rechnung kennt. In Wirklichkeit besitzt der Kunde ein Datenfeld mit Referenzen auf Rechnungsobjekte. Sie können also ein Kundenobjekt nach seinen Rechnungen oder gezielt nach einer besonderen Rechnung fragen. Eine Rechnung hat ein Datum, einen Gesamtpreis und besteht wiederum aus mindestens einer Rechnungsposition. Eine Rechnungsposition hat ihrerseits stets einen Bezug zu genau einem Artikel. Von diesem Artikel wird eine bestimmte Menge eingekauft. Ein Artikel kann auf mehreren Rechnungspositionen erscheinen. In der Datenbankmodellierung entspricht dies dem in Abbildung 3.20 dargestellten ER-Diagramm.

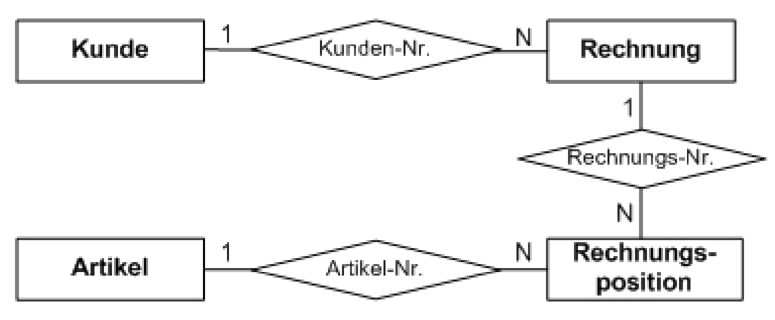


Abbildung 3.20: ER-Diagramm der Beziehungen der DB-Tabellen

Zusammensetzen: Aggregation und Komposition

Wenn Objekte von zwei Klassen sich gegenseitig kennen können, ist dies eine lose Beziehung zwischen den beiden Klassen. Beide Klassen könnten auch unabhängig voneinander existieren. Eine sehr enge Beziehung zwischen zwei Klassen besteht, wenn die eine Klasse von der anderen vererbt wurde. Dann *ist ein* Objekt der Unterklasse ja gleichzeitig auch ein Objekt der Oberklasse.

Eine Bindung zwischen zwei Klassen, die nicht gerade so stark ist wie bei der Vererbung, wird bei einer *Aggregation* aufgebaut. Diese Beziehung kann man mit der Phrase *besteht aus* bzw. *ist Teil von* beschreiben und sie wird dann angewendet, wenn ein Objekt aus einem oder mehreren anderen Objekten zusammengesetzt ist.

Betrachten wir das Beispiel der Kunden, Rechnungen, Rechnungspositionen und Artikel aus den Abbildungen 3.19 und 3.20. Selbst wenn ein Kunde nur dann ein Kunde ist, wenn er die erste Rechnung erhält, besteht ein Kunde nicht aus der Rechnung. Ebenso kann man eine Rechnung fragen, zu welchem Kunden sie gehört, jedoch ist der Kunde nicht als Ganzes Bestandteil der Rechnung.

Eine Rechnung ohne eine einzige Rechnungsposition macht jedoch keinen Sinn. Auf dem Dokument der Rechnung sind eine oder mehrere Positionen abgedruckt. Die Rechnungsposition ist demnach ein essenzieller Bestandteil der Rechnung. Ein Artikel ist unabhängig von einer Rechnungsposition, aber eine Rechnungsposition besteht immer genau aus einem Artikel. Auch hier kann man also von einer Aggregation sprechen.

Eine *Komposition* ist eine stärkere Bindung des Teils an sein Ganzes. Ein Unterschied zur Aggregation ist, dass die Existenz des Teilobjekts durch die des übergeordneten Objekts bedingt ist. Das Teilobjekt kann also nicht allein existieren. Im Gegensatz zur Aggregation kann bei einer Komposition das Teil zu einem Zeitpunkt nur zu genau einem Ganzen zugehörig sein. Wird das Ganze gelöscht, müssen auch dessen Teilobjekte gelöscht werden, die zu diesem Zeitpunkt Bestandteil des Ganzen waren. Sie müssen für die Realisierung einer Komposition die Lebensdauer der Objekte verwalten. Im Vergleich zu einer Aggregation sind Kompositionen wesentlich seltener.

Im Beispiel der Rechnungsverwaltung macht eine einzelne Rechnungsposition ohne eine zugehörige Rechnung keinen Sinn. Ebenso ist eine Rechnungsposition nicht gleichzeitig mehreren Rechnungen zugeordnet. Man kann hier also von einer Komposition sprechen. Ein Artikel kann aber auch existieren, ohne dass er zu einer Rechnungsposition gehört. Wenn mehrere Kunden denselben Artikel erwerben, so erscheint er in mehreren Rechnungspositionen. Hier sollte also eine Aggregation gewählt werden.

Abstrakte Klassen und Interfaces

Bereits im Beispiel des Autohauses wurde eine Oberklasse *Person* definiert, von der sich über mehrere Stufen der Vererbung die Klasse *Verkäufer* und *Kunde* abgeleitet haben. Auf eine ähnliche Art wurden von der Oberklasse *Tier* die Klassen *Hund* und *Katze* abgeleitet. Zusätzlich wurde gesagt, dass man Objekte aus Klassen mit dem Befehl *new* erstellen kann.

Es stellt sich aber die Frage, warum man Personen in der Anwendung anlegen kann, die weder Kunden noch Lieferanten oder Mitarbeiter sind. Weitere Personen machen im

System keinen Sinn. Ebenso möchten Sie verschiedene Tiere in Ihrer Anwendung abspeichern. Wenn Sie jedoch ein Objekt der Klasse *Tier* anlegen, ist dies zwar ein Tier, aber weder ein Hund noch eine Katze. Es wurde auch bereits eine Methode *gibLaut* definiert, die jedes Tier haben soll. Was ist jedoch der Default-Laut eines Tiers? Es ist nicht sehr praktikabel, einen solchen Standardlaut zu verwenden.

Für diesen Fall können Klassen als *abstrakt* definiert werden. Von einer abstrakten Klasse kann man zwar vererben, jedoch keine Objekte erstellen. Der *new*-Befehl kann also nicht angewendet werden. In einer Klasse können Sie ebenso einzelne Methoden mit ihren Ein- und Ausgabeparametern deklarieren, ohne jedoch den Inhalt der Methode auszuprogrammieren. Wenn Sie von dieser Klasse vererben, wird die abstrakte Methode mit vererbt. Wenn Sie von der Kindklasse dann konkrete Objekte erstellen wollen, sind Sie gezwungen, die deklarierten abstrakten Methoden auszuprogrammieren.

Ein *Interface* können Sie insofern mit einer abstrakten Klasse vergleichen, als Sie auch von einem Interface keine Objekte anlegen können. Bereits zu Beginn der Einführung in die Objektorientierung wurde beschrieben, dass die Eigenschaften einer Klasse ausschließlich über Methoden zugreifbar sein sollen. Als Schlussfolgerung interessiert Sie die interne Verwaltung der Eigenschaften nicht. Das Einzige, was Sie als Anwender einer fremden Klasse wissen müssen, sind die Namen der Methoden und deren Ein- und Ausgabe-Parameter. In einem Interface deklarieren Sie reine Funktionalität, also Methoden mit deren Parametern. Eine Klasse, die das Interface implementiert, muss die dort deklarierten Methoden dann ausprogrammiert haben. Eine Klasse kann mehrere Interfaces implementieren.

Eine zweite Klasse kann eine Referenz auf das Interface besitzen und dann ein Objekt der Klasse anlegen, die das Interface implementiert. Über die Referenz auf das Interface können die dort deklarierten Methoden dann verwendet werden. Der besondere Vorteil von Interfaces liegt darin, dass Sie die Interfacespezifikation im Klartext an verschiedene Entwicklerteams oder auch an Ihre Kunden offenlegen können. Sie können das Interface dann nutzen, um ihre eigene Software an Ihre Anwendung anzudocken. Interfaces fördern also die Wiederverwendbarkeit sowie das Komponentendenken, bei dem eine Anwendung aus verschiedenen Modulen möglichst einfach zusammengesetzt werden kann.

Klassenattribute und Klassenmethoden

Bislang gehörte jede Eigenschaft zu jedem Objekt einer Klasse; jeder Kunde hat einen Namen und jede Rechnung eine Anzahl von Rechnungspositionen. Es ist jedoch auch möglich, eine Eigenschaft der Klasse selbst zuzuordnen. Dies kann beispielsweise für statistische Zwecke sinnvoll sein: Nehmen wir an, Sie wollen die Anzahl der Objekte, die von einer Klasse erzeugt worden sind, zählen. Ein einzelnes Objekt hat jedoch keine Kenntnis von der Existenz anderer Objekte. Da die Klasse der Bauplan für Objekte ist, liegt die Kompetenz hier bei der Klasse selbst. Ebenso können Sie die Frage nach der Anzahl der erzeugten Objekte auch dann stellen, wenn noch gar kein Objekt dieser Klasse erzeugt wurde. Hier bleibt Ihnen keine andere Wahl, als die Klasse selbst zu fragen. Eine solche Eigenschaft, die zur Klasse selbst gehört, wird als Klassenattribut oder auch als statisches Attribut bezeichnet.

Bei jedem Aufruf des Konstruktors kann die Anzahl der erzeugten Objekte inkrementiert und bei jedem Aufruf des Destruktors dekrementiert werden.

Für Klassenattribute gelten dieselben Regeln zur Sichtbarkeit wie für normale Attribute: Sie sollten nach Möglichkeit *private* deklariert sein. Um auf das Attribut zugreifen zu können, benötigen Sie eine entsprechende Klassenmethode, die ebenfalls statisch deklariert wird. Diese Methode können Sie auf der Klasse selbst anwenden; selbst dann, wenn noch kein Objekt dieser Klasse erzeugt worden ist.

Profitipp

Eine gute objektorientierte Modellierung verzichtet auf eine große Anzahl von Klassenattributen und Klassenmethoden und setzt stattdessen Verwaltungsklassen ein. Dies ist besonders bei größeren statistischen Auswertungen sinnvoll und bietet einen zentralen Zugangspunkt für die Objekte. So kann für die Klasse *Kunden* eine separate Klasse *Kundenverwaltung* erzeugt werden, die eine Liste aller erzeugten Kunden verwaltet. Diese kann dann beispielsweise auch Suchfunktionen anbieten.

3.2.3 Vom Geschäftsprozess zur objektorientierten Analyse

Wenn die Klassen, deren Eigenschaften und Methoden sowie die Abläufe in der Anwendung bekannt sind, ist die objektorientierte Implementierung relativ leicht durchführbar. Bis dahin ist jedoch ein weiter Weg zu gehen, insbesondere bei sehr großen Projekten. Alle bislang vorgestellten Vorgehensmodelle teilen diesen Prozess in eine Anforderungserhebung, eine fachliche Analyse der Anforderungen, ein technisches formales Design der Anwendung mit anschließender Implementierung. Der Rational Unified Process (RUP) definiert die Geschäftsprozeßanalyse (GPA), dessen Modellierung (GPM), die objektorientierte fachliche Analyse (OOA), das technische Design (OOD) und die Implementierung (OOP) in zeitlich stark verzahnte Phasen. Dort wird bereits gezeigt, dass die Implementierung gerade in großen Projekten nur ca. 20 % des Gesamtaufwands ausmacht, während die anderen Phasen incl. der Managementaufgaben mit ca. 80 % ins Gewicht fallen.

In diesem Kapitel werden nun neue Methoden vorgestellt, die von der Geschäftsprozessanalyse bis zum Ende der objektorientierten Analyse behilflich sein sollen. Hier werden insbesondere die folgenden Schritte durchgeführt:

1. Sprechen Sie mit dem Kunden über die „Wirklichkeit“, um sich die Sprache seiner Fachdomäne anzueignen
2. Stellen Sie fest, welche Arbeitsabläufe existieren und welche Ausschnitte der Arbeitsabläufe in die zu erstellende Anwendung abgebildet werden sollen (GPA/GPM)
3. Modellieren Sie fachlich, welche Objekte daran beteiligt sind und abstrahieren Sie daraus die Klassen (OOA)
4. Stellen Sie das fachliche Modell dem Kunden in seiner Sprache vor und verifizieren Sie die Korrektheit des Modells

Mit den Jahren hat sich herausgestellt, dass eine Einteilung in Phasen, die zu zuvor bestimmten Zeiten enden, nicht realistisch ist. Die Erstellung von Prototypen ist ebenso

immer stärker in den Vordergrund gerückt wie die Einbeziehung der Kunden und der zukünftigen Anwender. Auf dieser Basis sind die agilen Methoden im Gegensatz zu den dokumentenlastigen und bürokratischen Vorgehensmodellen entstanden, die man als „schwergewichtig“ bezeichnet.

Agile Methoden

Ein entscheidender Kritikpunkt an den traditionellen Verfahren zur Softwareentwicklung ist die fehlende Flexibilität. Die Praxis zeigt, dass selbst die Anforderungen einem ständigen Wandel unterworfen sind. Im Jahre 2001 wurde das „Agile Manifest“ mit grundlegenden Leitsätzen zur Softwareentwicklung formuliert. Agile Methoden sind konkrete Verfahrensweisen während der Softwareentwicklung, die sich auf die festgelegten Werte und Prinzipien stützen. Die Leitsätze besagen, dass die Menschen und deren Kommunikation, eine funktionierende Software und die Reaktion auf Änderungen wichtiger sind als Werkzeuge, Dokumentation, Vertragsverhandlungen und Verfolgung eines wohl ausgearbeiteten Plans. Agile Projekte sind also schwerer im Vorfeld kalkulierbar.

Die Entwicklung funktionierender Software erfolgt in mehreren kurzen, aufeinanderfolgenden Iterationen, in die die gesamte Konstruktionsphase aufgeteilt ist. Die klassischen Phasen in der Softwareentwicklung von der Modellierung bis zur Entwicklung sind Bestandteil jeder Iteration.

Auf diese Weise wird die Software iterativ und inkrementell entwickelt. Iterativ bedeutet die Zerlegung der Entwicklung in mehrere gleichartige Schritte. Jede Iteration erzeugt ein funktionsfähiges Teilergebnis. Inkrementell bedeutet, dass die Gesamtfunktionalität des Systems mit jeder Iteration wächst. Dadurch wird die Software schnell an den Kunden ausgeliefert, der unmittelbar Feedback erzeugt.

Im Folgenden werden einige dieser Methoden und Techniken vorgestellt, die sich durch alle Phasen der Softwareentwicklung ziehen. Sie können diese Techniken unabhängig voneinander einsetzen und ausprobieren mit dem Ziel, die Wünsche Ihres Kunden besser zu erfüllen.

Die Erstellung von Story Cards

Das erste Hilfsmittel, um sich den Wünschen des Kunden zu nähern, ist das gemeinsame Erstellen von Story Cards, die jeweils eine User Story enthalten. Dies geschieht zumeist in Workshops. Eine User Story ist eine in Alltagssprache textuell formulierte Softwareanforderung. Sie ist bewusst kurz gehalten und umfasst nicht mehr als zwei Sätze. Die Story Card ist die wichtigste Methode, um ein agiles Projekt zu steuern, denn aus den Story Cards entwickeln sich in kooperativer Zusammenarbeit mit dem Kunden die Anwendungsfälle, die die Funktionalität der zu erstellenden Anwendung in UML abbilden. Eine Story Card besteht in der Regel aus folgenden Feldern:

- Datum und fortlaufende Nummer
- Nummer der übergeordneten Story Card
- Aktivität: neu/Fehler beheben/Funktion erweitern/Funktion umgesetzt

- Priorität des Kunden und des Entwicklers
- Risiko- und Aufwandschätzung
- kurze, präzise Beschreibung
- Notizen
- aktueller Fortschritt: Datum, Status, Aufgabe, Kommentar

Abbildung 3.21 zeigt eine Story Card für die Erstellung einer Anwendung zur Verwaltung von Seminaren. Diese Karte zeigt eine Managementsicht auf die zu erstellende Software und dient dazu, sich die Fachsprache des Kunden anzueignen und die Aufgabe zu umreißen. Es müssen natürlich noch weitere Karten erstellt werden, die die einzelnen Funktionen der Seminarverwaltung auflisten und beschreiben.

Nr.: 023 **Übergeordnet: ---** **Datum: 18.10.2008**

Aktivität: **neu** [X] **Fehler beheben** [ ] **erweitern** [ ] **umgesetzt** [ ]

Risk/Value: **Kunde** [ ] **Progr.** [ ] **Risiko** [ ] **Zeitdauer** [ ]

Beschreibung: Seminare müssen mit Agenda und Preis angelegt und verwaltet werden. Ein *offenes* Seminar hat beliebig viele Termine. Jedem Termin ist Raum und Dozent zugeordnet. Es gibt eine min. und max. TN-Zahl.
Ein *Inhouse*-Seminar wird dagegen dyn. per Telefon verwaltet, da gibts keine festen Termine. Telefonate sind Vorgänge in der Kundenverwaltung.

Notizen: Funktion schon teilweise im Visual Basic Programm vorhanden. Kundenverwaltung wird noch im „alten“ Cobol-Programm ausgeführt.

Fortschritt:

Datum	Aufgabe	Status	Kommentar
04.12.	Übersicht	erledigt	Großteil der Funktionen schon im Visual Basic Programm enthalten.

Abbildung 3.21: Exemplarische Story Card

Risk/Value-Priorisierung und Planning Poker

Bereits auf der Story Card erscheinen Felder für die Risk/Value-Priorisierung. Bei der Bewertung einer geforderten Funktionalität kann man folgendermaßen vorgehen: Zunächst beurteilt Ihr Kunde oder ein zukünftiger Anwender die Wichtigkeit dieser Funktion auf einer Skala von 1 (unwichtig) bis 10 (unbedingt für den Betrieb notwendig).

Die Entwickler beurteilen im zweiten Schritt die Integration der Funktionalität im Kontext der Gesamtanwendung. Handelt es sich um eine Kernfunktion, von der viele andere Komponenten abhängen – Vergabe von 10 Punkten – oder ist dies lediglich eine Randfunktionalität, die an die Anwendung beliebig an- und abgekoppelt werden kann? Im zweiten Fall wird nur 1 Punkt vergeben.

Im dritten Schritt sind nochmals die Entwickler gefragt. Sie sollen nun die Schwierigkeit der technischen Umsetzung der zu erstellenden Funktionalität beurteilen. Ist die Funktionalität so kritisch, dass bei einer aktuellen Fehleinschätzung das gesamte Design überarbeitet werden muss – 10 Punkte – oder hat die geforderte Funktionalität kaum Auswirkungen auf das Gesamtdesign der Anwendung (1 Punkt)?

Abschließend wird beurteilt, wie (zeit-)aufwändig die Umsetzung der neuen Funktionalität ist. Dies wird meist vom Projektmanagement durchgeführt. Mögliche Techniken zur Aufwandschätzung sind die erwarteten Zeilen an Quellcode, eine Function-Point-Analyse zur Größenbestimmung der Anforderungen oder ein Schätzverfahren wie COCOMO (COnstructive COst MOdel) zur Kosten- und Aufwandschätzung.

Es ist verständlich, dass diese Einschätzungen nur dann aussagekräftig sind, wenn alle Beteiligten bereits über eine große Erfahrung in der Erstellung von Softwareprojekten und Informationen von bereits erfolgreich durchgeführten Projekten ähnlicher Größe verfügen.

Abbildung 3.22 zeigt die Aufspaltung der Anwendung zur Seminarverwaltung in einzelne Funktionen sowie die Priorisierung des Kunden (K), der Entwickler (E), die Einschätzung der technischen Schwierigkeit (S) und der geschätzte Aufwand. Die Priorisierungen des Kunden und der Entwickler werden in der Zielspalte addiert.

	Prio. Kunde (K)	Prio. Entwickler (E)	techn. Schwierigkeit (S)	Aufwand	Summe (K+E/S)	
neues Seminar anlegen	9	9	7	9	18/7	
Termin(e) anlegen	9	8	2	4	17/2	
zum Seminar anmelden	9	7	5	6	16/5	
Rechnung zum Seminar drucken	2	1	6	8	3/8	**-> WaWi**
Dozenten verwalten	3	5	3	2	8/3	
Kunden verwalten	9	7	8	8	16/8	

Abbildung 3.22: Priorisierung von Funktionalität

Als Ergebnis ist zu nennen, dass die Anwendung zur Kundenverwaltung und das Anlegen eines neuen Seminars sowohl von unserem Kunden als auch von den Entwicklern als bedeutend und aufwändig zu realisieren eingeschätzt wurde. Interessant ist auch die Rechnungsverwaltung, die momentan kaum relevant ist, jedoch schwierig zu realisieren. Die Rechnungsverwaltung wurde in diesem Beispiel in ein externes, zugekauftes Warenwirtschaftssystem ausgelagert. Dies ist leicht bedienbar und wäre nur mit hohem Kostenaufwand selbst zu implementieren gewesen.

Eine Frage besteht darin, wie feingranular die Aufspaltung der benötigten Funktionen erfolgen soll? Eine Aussage besteht darin, eine Aufteilung bis zu einer Scrum-Aufgabe (nächstes Kapitel) vorzunehmen, deren Lösung ca. 16 Mannstunden in Anspruch nehmen soll. Dies ist ggf. eine zu feine Gliederung; in der üblichen Praxis kann eine Aufteilung in eine Mannwoche, also ca. 40 Mannstunden, erfolgen.

Werden die Prioritäten von dem Kunden und von den Entwicklern als „hoch" eingestuft, so ist die benötigte Funktionalität für die Gesamtanwendung wertvoll. Schwierige Funktionen implizieren ein hohes Risiko für die erfolgreiche Implementierung. Abbildung 3.23 stellt den Wert und das Risiko in einer Matrix gegenüber.

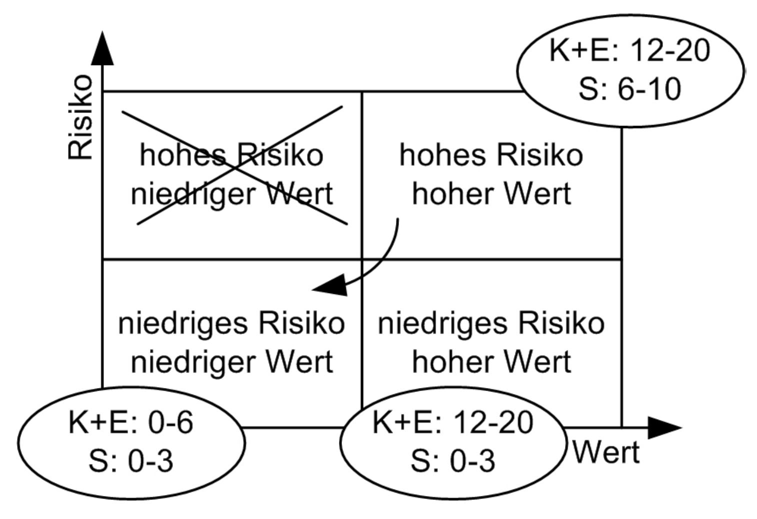


Abbildung 3.23: Risk/Value-Priorisierung

Auf den ersten Blick kann man denken, dass es sinnvoll ist, zunächst Funktionen mit niedrigem Risiko und hohem Wert zu implementieren, denn dadurch kann man in frühen Prototypen den Kunden zufriedenstellen. Bei größeren Projekten besteht hier jedoch die Gefahr, dass man relativ früh viele problemlose Module implementiert und im Anschluss daran wichtige Funktionen mit hohem Risiko implementieren muss. Währenddessen ist eine Änderung am Systemdesign wahrscheinlich, sodass die bestehenden Funktionen unter Umständen angepasst werden müssen.

Man sollte also zu Projektbeginn die Funktionen implementieren, die hohes Risiko und einen hohen Wert auf sich vereinigen. Im Anschluss daran sie die Funktionen mit niedrigem Risiko leicht realisierbar und zwar so weit, wie Zeit und Budget für das Projekt vor-

handen ist. Es macht keinen Sinn, Funktionen mit hohem Risiko und niedrigen Wert zu implementieren.

Die Risk/Value-Priorisierung wird nicht einmalig, sondern nach jeder Projektiteration durchgeführt. Die Prioritäten können sich dabei mit der Zeit verlagern und es können neue Funktionen hinzukommen.

Die Abschätzung und die Entscheidung, welche Funktionen in der nächsten Iteration realisiert werden, werden zusammen mit den Stakeholdern getroffen. Dieses Meeting wird als Planning Game oder Planning Poker bezeichnet.

Scrum

Bei Scrum (engl. „das Gedränge“) handelt es sich um ein agiles Vorgehensmodell, das dem Prinzip der schlanken Produktion (Lean Production) folgt. Die Begründung für Scrum liegt darin, dass die Softwareentwicklung sehr komplex ist und daher im Voraus weder in große abgeschlossene Phasen noch in einzelne Arbeitsschritte unterteilbar. Scrum lehnt also generell die Planung der Softwareentwicklung im Vorfeld ab. Stattdessen sollen die Teammitglieder ihre Arbeit weitgehend selbst organisieren. Dies geht soweit, dass die Entwickler auch die eingesetzten Entwicklungswerkzeuge und -methoden selbst wählen können, was in größeren Projekten umstritten ist.

Das zentrale Element bei Scrum ist der Sprint. Dabei handelt es sich um die Umsetzung einer Iteration, die ca. 30 Tage dauern soll. Vor dem Sprint werden die Produktanforderungen des Kunden in einem Produkt-Backlog gesammelt. Diese Liste beinhaltet alle Funktionalitäten, die der Kunde wünscht, inklusive einer Priorisierung der Funktionen, wie sie aus einer Risk/Value-Analyse stammen kann.

Hoch priorisierte Features werden im Aufwand geschätzt und in das so genannte Sprint-Backlog übernehmen. Diese Liste enthält alle Aufgaben, um das Ziel des Sprints zu erfüllen. Eine Aufgabe soll dabei in nicht mehr als 16 Stunden realisierbar sein. Längere Aufgaben sollten in Teilaufgaben zerlegt werden.

Während eines Sprints findet ein tägliches kurzes Scrum-Meeting statt, bei dem sich das Team gegenseitig die folgenden Fragen stellt:

- Bist du gestern mit dem fertig geworden, was du dir vorgenommen hast?
- Welche Aufgaben wirst du bis zum nächsten Meeting bearbeiten?
- Gibt es ein Problem, das dich blockiert?

Nach einem Sprint wird das Ergebnis einem informellen Review unterzogen, an dem sowohl das Entwicklerteam als auch Vertreter der Kunden- und Benutzerseite teilnehmen. Bei dem Ergebnis handelt es sich stets um einen funktionstüchtigen Prototyp der bislang erstellten Anwendung. Dieser Prototyp wächst evolutionär.

Der Kunde und die zukünftigen Anwender prüfen, inwiefern das Sprintergebnis den Anforderungen entspricht. Änderungswünsche werden wieder in das Produkt-Backlog aufgenommen und die nächste Iteration kann beginnen. Abbildung 3.24 fasst den Ablauf eines Scrum-Projekts zusammen.

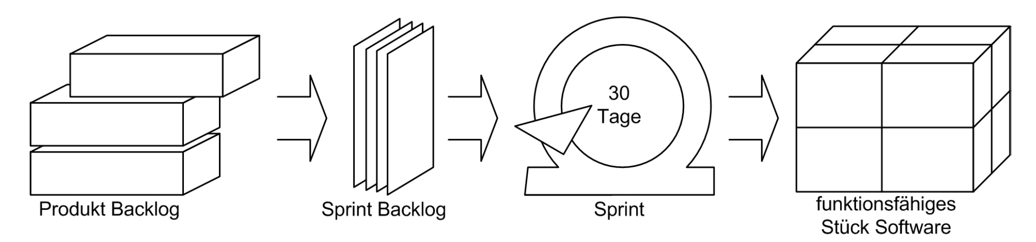


Abbildung 3.24: Ablauf eines Scrum-Projekts

3.2.4 Von der Analyse zum objektorientierten Design

Die im vorigen Kapitel skizzierten Methoden tragen dazu bei, aus noch unklaren Anforderungen des Kunden ein fachliches Modell der geforderten Funktionalität zu erzeugen. Scrum ist eine Methodik, die sich durch den gesamten Entwicklungszyklus der Anwendung zieht.

In diesem Kapitel werden nun Hilfsmittel vorgestellt, wie man aus dem fachlichen Modell eine technische Beschreibung erstellt. In der Objektorientierung bedeutet dies im Wesentlichen, die beteiligten Klassen mit deren Eigenschaften und Methoden zu ermitteln. Erst danach kommt PHP im Rahmen der objektorientierten Programmierung zum Einsatz.

Bei der Einführung in die Objektorientierung wurden die Klassen mit deren Eigenschaften und Methoden „nach Gefühl" des Analytikers ermittelt. In der Praxis ist dies tatsächlich die gängige Vorgehensweise. Es gibt jedoch einige Heuristiken, die Ihnen bei der Modellierung behilflich sein können.

Die Verb-/Substantiv-Methode

Die Verb-/Substantiv-Methode dient zur Bestimmung von Klassen, Eigenschaften und Methoden aus einer textuellen Problembeschreibung, beispielsweise aus Anforderungsbeschreibungen. Die Methode kann aber – wenn Sie als Analytiker darauf trainiert sind – auch in Kundengesprächen zum Einsatz kommen. Aus dem folgenden, bereits zusammengefassten Beispiel sollen die notwendigen Hauptklassen, Eigenschaften und Methoden extrahiert werden:

Beispiel

Es soll eine PHP-Anwendung zur Verwaltung von Studenten und Übungen erstellt werden. Eine Übung besteht aus maximal 10 Übungsgruppen, zu denen der Raum und die Uhrzeit gespeichert ist. Jeder Raum hat eine Raumnummer und eine bestimmte Anzahl von Plätzen. Für jeden Studenten wird Name, Matrikelnummer und E-Mail-Adresse erfasst. Ein Student kann für eine der Gruppen angemeldet sein. In einer Gruppe ist die Zahl der angemeldeten Studenten nur durch die Zahl der Plätze limitiert.

Jedes Substantiv ist prinzipiell ein Klassenkandidat oder ein Attributkandidat. Ein Substantiv mit einem einfachen Wert, wie eine einzelne Zahl oder Zeichenkette, ist ein Attributkandidat.

Um nun von den Kandidaten zu den Klassen zu gelangen, werden zunächst doppelt vorkommende Kandidaten gestrichen und stets die Singularform des Substantivs verwendet. Dadurch bleiben folgende Klassenkandidaten bestehen:

- PHP-Anwendung
- Verwaltung
- Student
- Übung
- Raum
- Übungsgruppe

Nun werden die Substantive gestrichen, die zum Beschreibungstext, aber nicht zum Problem gehören. So muss die PHP-Anwendung sicherlich nicht als eigene Klasse deklariert werden. Der Name *Verwaltung* deutet auf eine separate Verwaltungsklasse für die Übungen. Die anderen vier Klassen bilden den Kern der Fachlogik.

Es wird gesagt, dass „jede Übung aus maximal 10 Übungsgruppen besteht“. Die Phrase *besteht aus* deutet auf eine Aggregation oder eine Komposition. Da eine Übungsgruppe genau zu einer Übung gehört und allein keinen Sinn macht, ist hier eine Komposition zwischen den beiden Klassen zu erstellen. Eine Übung verwaltet eine Liste mit ihren Gruppen, die bis zu 10 Referenzen auf Gruppenobjekte enthalten kann.

Zu jeder Gruppe werden *Raum* und *Uhrzeit* gespeichert. Zwischen den Klassen *Gruppe* und *Raum* existiert also eine Assoziation. Die Uhrzeit ist eine Eigenschaft der Übungsgruppe. Durch die Aussage, dass jeder „Raum eine Raumnummer und eine bestimmte Anzahl von Plätzen“ hat, werden die Eigenschaften *Raumnummer* und *Plätze* als elementare Datentypen festgelegt. Die Eigenschaften eines Studenten werden durch die Aussage festgelegt, dass für „jeden Studenten Name, Matrikelnummer und E-Mail-Adresse erfasst“ werden.

Über die Anmeldung kann eine Assoziation zwischen einem Studenten und einer Übungsgruppe entstehen. Da jede Übungsgruppe ihren Raum kennt, kennt sich auch die darin gespeicherte maximale Anzahl an Plätzen. Wenn sich ein Student zu einer Gruppe anmelden will, so kann die Übung über die Raumreferenz prüfen, ob noch ein Platz frei ist.

Verben können Hinweise auf Methoden geben. In diesem Text sind jedoch nur wenige Hinweise auf Methoden zu finden. Die Erfassung eines Studenten mit dessen Namen, Matrikelnummer und E-Mail-Adresse deutet auf die Parameter des Konstruktors, die bei der Objekterzeugung notwendig sind.

Zusätzlich ist das Anmelden an eine Übungsgruppe zu nennen. In der Modellierung besitzt eine Übungsgruppe dann eine Methode *anmelden*, bei der als Parameter ein Studentobjekt übergeben wird.

Die Methode der CRC-Karten

Auch die CRC-Karte (Class-Responsibility-Collaboration-Karte) ist ein Hilfsmittel für das objektorientierte Design. Das Prinzip besteht darin, für jede Klasse eine Karteikarte zu erstellen und auf ihr deren Eigenschaften zu notieren. Eine allgemeine Notation besteht aus den folgenden Bereichen:

- Oben steht der Name der Klasse und ggf. deren Oberklasse.
- Auf der linken Seite schreibt man die Aufgaben, die die Klasse erfüllen soll in kurzen, präzisen Stichpunkten.
- Auf der rechten Seite stehen die Klassen, mit denen die beschriebene Klasse zusammen arbeitet.
- Auf der Rückseite beschreibt man die Klasse etwas detaillierter anhand eines Verzeichnisses der Methoden und der Eigenschaften.

Der Vorteil der CRC-Karten liegt in der einfachen Handhabung. Man kann problemlos Informationen hinzufügen oder streichen. Auf Grund des einfachen Ansatzes ist man auch unabhängig von verwendeten Programmiersprachen und -werkzeugen. Der begrenzte Platz zwingt die Beteiligten zusätzlich dazu, sich auf die wesentlichen Aufgaben einer Klasse zu konzentrieren. Die CRC-Karten werden meist in kreativen Workshops erstellt, an denen Vertreter der Entwickler, des Managements des Kunden sowie zukünftige Anwender teilnehmen.

Assoziationen zwischen den Klassen kann man auf unterschiedlichen Wegen veranschaulichen. Entweder schreibt man die Namen der behandelten Klassen auf die Karte oder man befestigt die Karten an einer Wand und zeichnet Striche zwischen den Karten. Auf diese Weise müssen sich die Teilnehmer mehr bewegen, was die Situation auflockert. Die Atmosphäre in den Workshops soll generell ungezwungen und frei von Führungshierarchien sein.

Als Vorgehensweise kann man im ersten Schritt typische Anwendungsfälle der zukünftigen Software, wie die Erstellung eines Neukunden oder das Aufgeben einer Bestellung durchspielen. Währenddessen hält der Systemanalytiker auf den CRC-Karten neue Zuständigkeiten und Partnerklassen fest. Im Lauf der Zeit ergibt sich so ein vollständiges Bild.

Zwischendurch werden einzelne Klassen detaillierter betrachtet, deren Aufgabengebiete spezifiziert sowie die wichtigsten Eigenschaften und Methoden skizziert. Wichtig ist dabei, dass mit der Zeit möglichst alle typischen Anwendungsfälle diskutiert werden, da ansonsten Klassen übersehen werden könnten.

Abbildung 3.25 zeigt exemplarisch eine CRC-Karte der Klasse *Seminar* auf Managementebene. Als Übung können Sie sich überlegen, wie die CRC-Karten der Klassen *Termin*, *Raum*, *Dozent*, *Anmeldung* und *Teilnehmer* aussehen könnten.

Seminar
Aufgaben: Erstellen und verwalten von Seminaren und deren Inhalten. Man soll Seminare suchen können und im Internet veröffentlichen. Man kann sich anmelden, solange die max. TN-Tahl nicht erreicht ist. Abmelden geht auch jederzeit. Der Seminaranbieter kann notfalls ein Seminar auch stornieren, wenn keine Anmeldung vorliegt oder der geplante Dozent krank ist und kein Ersatz gefunden werden konnte. **Partnerklassen:** Seminarverwaltung, Rechnung, Termin -> (Raum, Dozent, Anmeldungen -> (Teilnehmer))

Attribute: ID, Dauer, Liste mit Terminen, Preis, Ort Kurzbeschreibung, Inhalt, Zielgruppe min. TN, max. TN., Liste mit Anmeldungen, storniert ja/nein **Methoden:** + Daten anzeigen () + Daten aktualisieren (Dauer, Preis, Ort, Kurzbeschreibung, Inhalt, Zielgruppe, min. TN, max. TN.) + Termin hinzufügen / ändern / löschen + anmelden (Termin T, Teilnehmer TN), + abmelden (Termin T, Teilnehmer TN) + stornieren (Termin T) {wenn kein Dozent verfügbar oder min. TN nicht erreicht} + entfallen (Termin T) {wenn keine TN vorhanden}

Abbildung 3.25: Exemplarische CRC-Karte der Seminarverwaltung, Vorder- und Rückseite

3.2.5 Objektorientierte Programmierung

Wie bereits bei der objektorientierten Analyse und Design, ziehen sich die agilen Methoden auch in die objektorientierte Programmierung hinein, in der das technische Konzept der Klassenstrukturen in PHP-Quellcode umgesetzt werden soll.

Testgetriebene Entwicklung

Als erstes Konzept, das immer größere Verbreitung findet, ist die testgetriebene Entwicklung (TDD: Test-driven Development) zu nennen. Bei der testgetriebenen Entwicklung erstellen Sie die Softwaretests konsequent vor den zu testenden Komponenten.

Zumeist werden die Tests unabhängig von der zu testenden Anwendung entwickelt oder sogar nachdem die Anwendung entwickelt wurde. Dies führt oft dazu, dass nicht die erforderliche Testabdeckung erzielt wird. Oft werden die Tests auch nur halbherzig durchgeführt und die Anwendung durchläuft beim Kunden zunächst eine „Betaphase", weil ein Auslieferungstermin eingehalten werden musste.

Wie aber können Sie die Tests vor den zu testenden Komponenten erstellen, wenn Sie nicht genau wissen, wohin Ihr Kunde die Entwicklung steuern wird? Wie müssen Sie

vorgehen, wenn Sie späte Änderungen der Anforderungen in das Testkonzept integrieren wollen?

Bei Entwicklern, die nicht testgetrieben entwickeln, hören Sie oft nach der Erstellung des Quellcodes die Formulierung: „Das Programm ist zu komplex. Man kann es nicht so einfach testen."

Der Lösungsansatz besteht auch hier in einem iterativ-inkrementellen Vorgehen. Der Design-, Test- und Entwicklungsprozess ist dem organischen Anpassungs- und Wachstumsprozess sehr ähnlich. Der Trend führt also zu evolutionärem Prototyping mit Phasen des Refactorings. Das testgetriebene Entwickeln ist eine Just-in-Time-Technik, um auf wechselnde Anforderungen flexibel einzugehen.

Das Testen einzelner Funktionalität auf Quellcodeebene – meist auf der Ebene einzelner Objekte – wird als Unit Testing bezeichnet. Die Unit-Tests und der zugehörige Quellcode werden dabei parallel zueinander in kleinen und wiederholten Mikroiterationen entwickelt. Die Dauer einer Iteration dauert nur wenige Minuten, um den Entwickler nicht von seiner Problemstellung abzulenken. Dennoch erfordert die testgetriebene Denkweise eine gewisse Selbstdisziplin und Umgewöhnung im Vergleich zum „Herunterhacken" von Quellcode.

Profitipp

Stellen Sie sich stets die Frage: Was soll die Funktion leisten, die ich jetzt programmieren will? Schreiben Sie die Antwort darauf direkt in ihren Quellcode! Denken Sie also wie jemand, der Ihre Funktion später anwenden wird und von der internen Realisierung keine Ahnung hat.

Eine Mikroiteration der Programmierung unter Berücksichtigung der testgetriebenen Entwicklung hat vier Hauptteile:

1. Schreiben Sie einen Test für das erwünschte fehlerfreie Verhalten der geplanten Methode. Wenn Sie bereits Fälle kennen, bei denen Ihre Methode fehlschlagen soll, so geben diese ebenfalls gültige Testfälle. Ein Beispiel ist das Anmelden eines Studenten an einer bereits gefüllten Übungsgruppe. Der Quellcode, um diese Tests erfolgreich auszuführen, existiert jedoch noch nicht.
2. Implementieren Sie den notwendigen Quellcode mit möglichst wenig Aufwand, bis alle bisher erstellten Tests bestanden werden.
3. Führen Sie ein Refactoring durch, um die Qualität des Quellcodes und des Gesamtdesigns zu verbessern. Entfernen Sie insbesondere Codeduplikate, Debug-Ausgaben und abstrahieren Sie das bereits bestehende Design, wo es notwendig ist.
4. Führen Sie nochmals alle bisher erstellten Tests aus. Werden sie weiterhin erfolgreich ausgeführt, können Sie den ersten Schritt mit der nächsten Funktionalität ausführen.

Die dazu erstellten Unit-Tests werden als Grey-Box-Tests bezeichnet, als Kompromiss zwischen einem Blackbox-Test und einem Whitebox-Test. Bei einem Blackbox-Test kennen Sie die interne Realisierung der zu testenden Anwendung nicht. Die Gemeinsamkeit zur testgetriebenen Entwicklung liegt darin, dass Sie hier den Test im Vorfeld program-

mieren, also die zu testende Anwendung noch gar nicht realisiert haben. Bei einem Whitebox-Test prüfen Sie alle möglichen Pfade durch ihre programmierten Anweisungen. Hier liegt die Gemeinsamkeit darin, dass Sie sowohl die erfolgreichen Ausgaben als auch typische Fehlerfälle berücksichtigen sollen.

Mittlerweile existieren Werkzeuge, die Sie bei der testgetriebenen Entwicklung unterstützen. So sammelt PHPUnit separat alle bisherigen Test und führt sie mit einem Mausklick aus. Die benötigte Ausführungszeit der Unit-Tests sollte einige Sekunden nicht überschreiten, um nicht von der Entwicklung abzulenken.

Der erstellte Bestand an Unit-Tests ist gleichzeitig eine Dokumentation der Anwendung auf Quellcodeebene. Sie erzeugen während der Entwicklung eine „ausführbare Spezifikation", indem automatisch definiert wird, was Ihre entwickelten Methoden leisten und in welchen Fällen Fehler ausgegeben werden.

Wenn Sie sich die testgetriebene Denkweise aneignen wollen, ist Abbildung 3.26 hilfreich. Zu testen sind die Methoden der Objekterzeugung für ein neues Seminar und das Anmelden für ein Seminar. Zu Beginn können Sie sich noch vor der Erstellung eines Tests Gedanken darüber machen, welche Bedingungen erfüllt sein müssen, damit der Testfall eintritt. Im nächsten Schritt müssen Sie sich die Zustände überlegen, die Sie im Erfolgsfall und im Fehlerfall erwarten. Gegebenenfalls müssen Sie daraus mehrere Unit-Tests erzeugen.

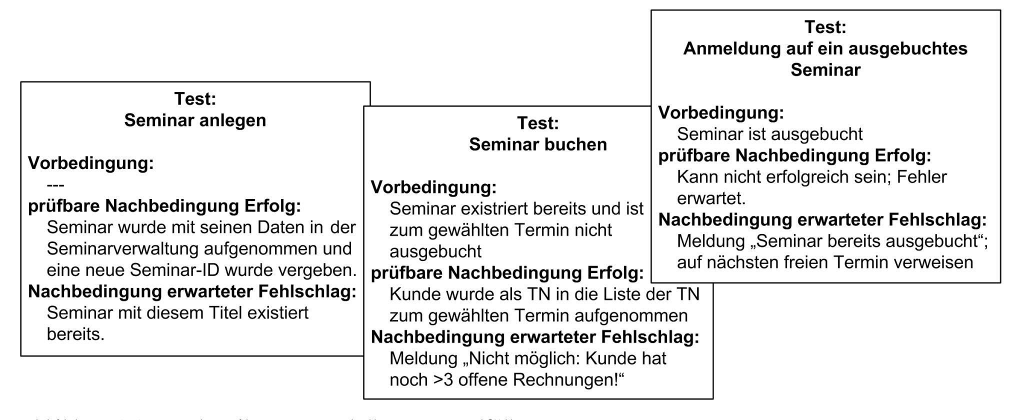


Abbildung 3.26: Vorbereitung zur Erstellung von Testfällen

Featuregetriebene Entwicklung

Mit Scrum wurde bereits eine agile Methode zur Softwareentwicklung vorgestellt. Das Produkt-Backlog definiert dort die gewünschte Funktionalität der zu erstellenden Software. Mit jeder Iteration wird das Produkt-Backlog aktualisiert und die nächsten Schritte der Entwickler im folgenden Sprint geplant.

Die featuregetriebene Entwicklung (FDD: Feature-driven Development) besitzt diese Rückkopplung nicht. Eine featuregetriebene Entwicklung ist weniger bürokratisch als Scrum, lässt jedoch auch weniger Feedback durch den Kunden zu. Diese Methode für ein agiles Projektmanagement wurde 1997 definiert, als ein zeitkritisches Projekt in 15

Monaten von einem relativ großen Entwicklerstamm von 50 Entwicklern umgesetzt werden sollte.

Die Idee besteht darin, dass Funktionalität das wichtigste Ergebnis ist, das der Kunde wünscht. Daher definieren die Fachexperten des Kunden und die Entwickler zunächst unter der Leitung eines Chefarchitekten den Inhalt und Umfang der zu entwickelnden Anwendung. Der Chefarchitekt spielt eine zentrale Rolle in diesem Modell, sowohl im Projektmanagement als auch in der Vermittlung zwischen Vertretern des Kunden und der Entwickler. In kleinen Gruppen werden im Folgenden fachliche Modelle für die einzelnen Bereiche der Anwendung erstellt. Das Ziel ist eine fachliche Einigung der Beteiligten.

In der zweiten Phase teilen erfahrene Entwickler die in der ersten Phase festgelegten fachlichen Teilmodelle in Features auf. Features in der Seminarverwaltung wären beispielsweise das Anlegen eines neuen Seminars oder das Buchen einer Anmeldung auf einen vorhandenen Seminartermin. Die entstehende Featureliste entspricht stets folgendem Schema:

- Aktion: Buchung
- Ergebnis: eine Anmeldung
- Objekt, auf dem die Aktion ausgeführt wird: ein vorhandener Seminartermin

Ähnlich wie die Aufgabenteilung bei Scrum sollte die Umsetzung eines Features nicht länger als zwei Wochen benötigen.

In der dritten Phase werden die Features vor allem von dem Projektleiter priorisiert. Dabei sollten auch die Abhängigkeiten zwischen den Features beachtet werden. Auch der Kunde kann seine Meinung hier mit einbringen, was beispielsweise über eine Risk/Value-Priorisierung geschehen kann.

Auf Basis der Featureliste werden die Fertigstellungstermine festgelegt und den Teamleitern der Entwickler zugeordnet. Zusätzlich können einzelnen Entwickler die Verantwortung für bestimmte Kernklassen – wie *Seminar* oder *Anmeldung* in der Seminarverwaltung – zugewiesen werden. So ergeben sich klare Verantwortlichkeiten.

Von großer Bedeutung ist, dass diese ersten drei Phasen sehr unbürokratisch und pragmatisch abgehandelt werden sollen. In dem zeitkritischen Beispielprojekt wurden diese drei Phasen innerhalb von wenigen Tagen abgehandelt. Bei dieser Vorgehensweise liegt das Verhältnis zwischen der Analyse und dem Design zu der eigentlichen Implementierung also nicht bei 4:1, wie es bei dem schwergewichtigen RUP geschildert wurde.

In der vierten Phase werden die technischen Modelle realisiert. Dazu gehören die Modellierung der einzelnen (Unter-)Klassen mit deren Eigenschaften und Methoden sowie die Spezifikation der technischen Abläufe. Diese können mit Datenflussdiagrammen oder Sequenzdiagrammen der UML (Kap. 3.2.6) erstellt werden. Die Entwickler implementieren erste Klassen- und Methodenrümpfe, die von den Teamleitern begutachtet werden. Bei Unklarheiten werden Fachexperten des Kunden herangezogen.

In der fünften und letzten Phase werden die Features ausprogrammiert. Dies erfolgt unter Verwendung von Unit-Tests und Pair-Reviews (nächstes Unterkapitel). Abbildung 3.27 fasst den Vorgang der 5 Phasen nochmals zusammen.

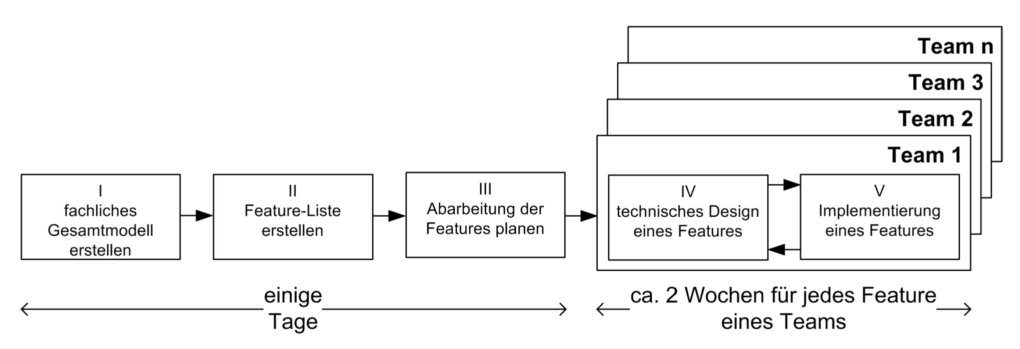


Abbildung 3.27: Vorbereitung zur Erstellung von Testfällen

Paarprogrammierung und Peer Review

Die hier vorgestellte agile Methodik betrifft, wie auch das testgetriebene Entwickeln, den Vorgang des Codings selbst. Die Paarprogrammierung kann mit anderen agilen Methoden kombiniert werden, wie mit der test- oder featuregetriebenen Entwicklung. Die zentrale Idee dabei ist, dass zwei Entwickler mit ähnlich großer Erfahrung an einem einzigen Arbeitsplatz sitzen, mit einer Tastatur und einem Bildschirm.

Zu jedem Zeitpunkt schreibt nur einer der beiden Entwickler den Quellcode. Dieser Entwickler wird als *Driver* bezeichnet; er hält das Steuer in der Hand. Der zweite Entwickler, der *Navigator*, behält den etwas entfernteren Überblick über die Entwicklung. Während der Paarprogrammierung herrscht jedoch keine Arbeitsteilung, die Rollen zwischen Driver und Navigator wechseln alle 15 bis 30 Minuten.

Das Mobiliar des Arbeitsplatzes muss natürlich entsprechend eingerichtet sein, damit die Entwickler ihre Rollen auch angenehm wahrnehmen und ihre Unterlagen ausbreiten können, dass beide jederzeit Einsicht nehmen können, ohne dass Chaos entsteht. Außerdem sollten sich maximal 2 bis 3 Paare in einem Raum befinden, da die Geräuschkulisse sonst unangenehm sein könnte.

Die Unterbrechungen während der Programmiersession sollten minimiert werden; insbesondere muss auf das Telefonieren und das Abrufen bzw. Beantworten von E-Mails verzichtet werden. Dies erfordert natürlich eine gewisse Selbstdisziplin. Circa alle 2 Stunden oder wenn sich beide Partner an einem Problem festgefahren haben, sollte eine Pause eingerichtet werden, die dann nach Möglichkeit nicht am Arbeitsplatz statt findet.

Ein traditionell orientiertes Unternehmen wird an dieser Stelle die Frage stellen, warum es zwei Entwickler bezahlen muss, obwohl nur einer arbeitet. Die Antwort liegt darin, dass das Entwickeln wesentlich mehr ist als reines Coding. Kennen Sie als Entwickler das Gefühl, vor einem Problem zu sitzen und absolut keine Lösung zu finden? Sie betrachten oft sehr lange den bereits erstellten Quellcode und sehen den Fehler nicht? Erst wenn ein Kollege dazukommt – der sich meist mit dem Problem gar nicht auskennt – und Sie mit diesem Kollegen über das Problem reden, findet sich die Lösung nach einigen Minuten. Dieser Vier-Augen-Effekt wird bei der Paarprogrammierung durch die Anwesenheit zweier Entwickler ausgenutzt und durch die ständige Präsenz auch das grundlegende Design des Quellcodes verbessert.

Auch die Paarzusammenstellung soll sich regelmäßig ändern, damit sich die Entwickler nicht zu sehr aneinander gewöhnen. Dies setzt natürlich einen genügend großen Entwicklerstamm voraus, bei dem Einzelne nicht zu spezialisiert auf ein Fachgebiet sind. Wenn dies der Fall ist, wird durch die Teamwechsel das Wissen über den Quellcode im Unternehmen verbreitet und der Effekt der unverzichtbaren „single Heads of Knowlegde“ bzw. der „Key-Entwickler“ verringert. Key-Entwickler sorgen dafür, dass der Projektfortschritt stillsteht bzw. das Wissen aus dem Unternehmen verschwindet, wenn sie in Urlaub oder krank sind oder gekündigt haben.

Neben der Erhöhung sozialer Kompetenz, der verbesserten Güte des Quellcodedesigns, einer spannenderen Zusammenarbeit als das einsame Kodieren vor einem PC ist noch das so genannte *Collective Code Ownership* zu nennen. Sind Einzelne für ihren Quellcode verantwortlich, so neigen sie dazu, resistent gegenüber veränderten Anforderungen und Kritik an ihrem Design zu werden. Dies führt zu einem blockierenden Verhalten bei Veränderungen.

Auch wenn das Management des Unternehmens die Paarprogrammierung explizit fördert, was für einen Erfolg dieser Methode unabdingbar ist, ist die Beurteilung der Leistung des Einzelnen schwieriger geworden. Denn letztlich erhält jeder Entwickler seine persönliche Entlohnung. Außerdem kommt es vor, das manche Personen – die ggf. über eine extrem hohe fachliche Kompetenz verfügen – nicht in dieses agile System integriert werden können. Hier ist die Einzelarbeit einem Zwang zum Team auf jeden Fall vorzuziehen, da ansonsten die Gesamtmoral gefährdet ist.

An dieser Stelle kann an die Stelle der Paarprogrammierung die aus wissenschaftlichen Veröffentlichungen stammende Methode des Pair Reviews treten. Dabei entwickelt und testet ein einzelner Entwickler den Code. Im Anschluss daran wird die fertige Version der Softwarekomponente – meist handelt es sich dabei um eine Version einer Klasse – einem zweiten Entwickler zur Verfügung gestellt, der bislang an der Entwicklung dieser Komponente nicht beteiligt war. Ziel ist es, dass dieser zweite Entwickler sich tief in die erstellte Komponente einarbeitet. Der Reviewer verfasst eine kurze Stellungnahme zum gewählten Design und dessen Umsetzung. Bei gröberen Fehlern kann auch eine Änderung veranlasst werden, obwohl die Unit-Tests fehlerfrei liefen. Auch hier führt das Vier-Augen-Prinzip mittelfristig zu einer höheren Softwarequalität. Um die Verantwortung des Reviewers deutlich zu machen, sollte sein Name nachweislich neben dem Entwickler als zweiter Verantwortlicher für diese Klasse eingetragen werden. Dadurch wird zusätzlich das Wissen über den Quellcode verbreitet, wenn auch nicht so stark wie bei der Paarprogrammierung.

Das Prinzip des Model-View-Controllers (MVC)

In Kapitel 3.1.3 wurde bereits der Aufbau einer 3-Schichten-Architektur mit einer Datenzugriffsschicht, einer Fachlogik und einer Präsentationsschicht vorgestellt. Kapitel 2.2 zeigte die PHP-Funktionen, die für den Zugriff auf eine MySQL-Datenbank notwendig sind. Eine zu der 3-Schichten-Architektur verwandte Systemarchitektur ist das Prinzip des Model-View-Controllers. Diese Architektur ist gerade in Verbindung mit objektorientierten Ansätzen zur Strukturierung komplexer Anwendungen weit verbreitet.

Das Ziel ist es, einen Rahmen für einen flexiblen Programmentwurf vorzugeben, der eine spätere Erweiterung der zu erstellenden Anwendung erleichtert und eine Wiederverwendbarkeit der einzelnen Komponenten ermöglicht.

Das Modell enthält die darzustellenden Daten zumeist in Form einer (relationalen) Datenbank. Zum Datenmodell gehören auch die PHP-Funktionen, die auf diese Daten zugreifen sollen.

Die Präsentation ist sowohl für die Darstellung der benötigten Daten aus dem Modell als auch für die Entgegennahme von Benutzerinteraktionen zuständig. Darzustellende Daten werden zumeist in HTML-Tabellen unter Verwendung von Style Sheets aufbereitet, während Benutzereingaben zum größten Teil aus HTML-Formularen bestehen, die ausgefüllt und zu einer PHP-Seite zur Auswertung weitergeleitet werden.

Diese PHP-Seite wird als Steuerung bezeichnet und verwaltet die Rückgaben von einer oder mehreren Präsentationen. Die eingegebenen Daten des Anwenders werden entgegengenommen, auf Gültigkeit geprüft und ausgewertet. Dabei greift die Steuerung auf das Modell zu und leitet zu der entsprechenden Präsentation für den Anwender weiter. In der Steuerung befinden sich auch die modellierten PHP-Klassen.

Abbildung 3.28 zeigt die Trennung der Schichten nach dem MVC-Prinzip unter Verwendung von PHP. Die Dateien *login.html* bzw. *login.php* sowie *ok.html* bzw. *ok.php* bilden die Präsentationsschicht auf Basis von clientseitigem Quellcode wie HTML, JavaScript, AJAX, CSS usw. Die *auswertung.php* enthält den Kern der PHP-Fachlogik und bildet die Steuerung der Login-Funktion. Sie verwaltet auch den Zugriff auf das Datenmodell. Die *zugriff.php* verwaltet intern den Zugriff auf den Datenbankserver, setzt SQL-Abfragen ab und gibt die Antworten an die Steuerung weiter.

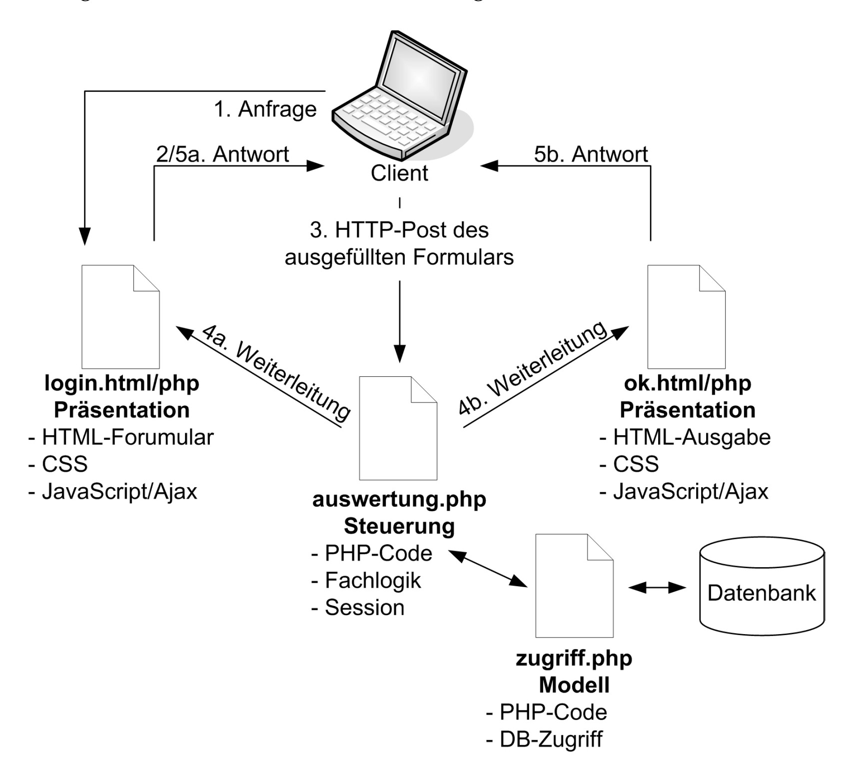


Abbildung 3.28: Das MVC-Prinzip in einer PHP-Realisierung

3.2.6 Die Bedeutung der Unified Modeling Language (UML)

Das Ziel einer objektorientierten Analyse, Design und Programmierung ist es, ein komplexes Softwaresystem zu beschreiben, wie

- die Verwaltung eines Autohauses
- eine Hotelverwaltung
- ein Buchungs- und Bestellsystem

Wenn Sie vom Kunden die geforderte Funktionalität in Erfahrung gebracht haben, die Klassen mit deren Eigenschaften und Methoden kennen sowie die geschäftlichen Abläufe, die in der Anwendung abgebildet werden sollen, dann können Sie mit der Implementierung beginnen. Es wurden bereits agile Methoden und Techniken vorgestellt, wie Sie an diese Informationen kommen. Es gibt bislang jedoch noch kein Mittel, um diese Erkenntnisse festzuhalten, zu dokumentieren und als schriftliche Diskussionsgrundlage zu verwenden.

Die UML (Unified Modeling Language) ist eine standardisierte, überwiegend grafische Sprache zur objektorientierten Modellierung von Systemen. Dabei muss es sich nicht unbedingt um eine Softwareanwendung handeln. Diese Sprache zieht sich von der Analyse über das Design bis zur Implementierung, begleitet Sie also durch den gesamten Prozess der objektorientierten Entwicklung. Die Sprache existiert seit 1994, die derzeit aktuelle Version lautet 2.1.

Die Sprache selbst wurde erfunden von Grady Booch, Ivar Jacobson und James Rumbaugh, die bei der Firma Rational Software angestellt waren. Sie haben auch das letzte schwergewichtige Modell zur Softwareentwicklung, den Rational Unified Process, spezifiziert. Die Weiterentwicklung und Standardisierung haben die drei Erfinder der Sprache an die OMG (Object Management Group) übergeben. Dieses Konsortium mit heute über 800 Mitgliedern ist international anerkannt für die herstellerunabhängige, systemübergreifende Standardisierung der Objektorientierung. Die OMG hat die UML dann 1997 als Standard akzeptiert und entscheidend zu der weltweiten Verbreitung der Notation beigetragen.

Die UML definiert eine Vielzahl von Diagrammtypen. Jeder Typ besitzt eine eigene Notation und stellt eine spezielle Sichtweise auf das modellierte System dar. Sie können einen Diagrammtyp mit einer Darstellung aus der Architektur beim Hausbau vergleichen. Eine Zeichnung mit einer Seitenansicht auf ein Haus zeigt sehr gut Treppenverläufe und die Höhe von Decken, jedoch kann man die Raumaufteilung nicht erkennen. Dies funktioniert besser mit einer Draufsicht. Ein Modell eines Hauses ist gut für Marketingzwecke geeignet, beispielsweise bei einer öffentlichen Ausschreibung. Einem solchen Modell sollten Sie aber nicht die Abmessungen für den realen Hausbau entnehmen; dies würde zu großen Messfehlern führen.

Jedes UML-Diagramm zeigt ebenso eine Sicht auf die zu erstellende Anwendung. Einige Aspekte können Sie an bestimmten Diagrammen besonders gut entdecken, andere Aspekte weniger gut. Dafür existieren dann wieder andere Diagramme. Klassen- und Paketdiagramme zeigen besonders gut die statische Struktur der zu erstellenden Anwendung. Dies betrifft die Datenhaltung und den Zusammenhang zwischen Klassen

und Modulen. Aktivitäts- und Sequenzdiagramme stellen dagegen insbesondere die Interaktion, Kommunikation und Abläufe in den Vordergrund und fokussieren die Dynamik in der zukünftigen Anwendung.

Profitipp

Sie können eine Anwendung niemals mit nur einem UML-Diagrammtyp beschreiben! Es gibt zwar wichtigere und unwichtigere Diagramme, aber wenn Sie die Vielfalt der Diagramme zu sehr eingrenzen, um nicht die gesamte Notation verwenden zu müssen, werden Sie auch einige Aspekte Ihrer Anwendung nicht betrachten!

Die Anwendung der UML eignet sich insbesondere bei der Programmierung im Großen, wenn also viele Stakeholder an dem Projekt beteiligt sind. Teile der UML können jedem Projektbeteiligten bekannt gemacht werden, sodass die Notation als gemeinsame Sprache und als Diskussionsgrundlage verwendet werden kann.

Der Detailgrad: Von der Wolke bis zur Muschel

Gerade zu Projektbeginn sind viele Anforderungen selbst dem Kunden noch nicht genau klar. Es kann daher nicht sofort eine fertige Spezifikation ausgearbeitet werden. Auch muss man bedenken, dass das Management des Kunden, das sich zu dem Projekt entschließt, aufgrund seiner Position im Unternehmen, aber auch aufgrund seiner Ausbildung ein völlig anderes Bild von der zukünftigen Anwendung hat als der zukünftiger Anwender, der ja ebenso bei Ihrem Kunden angestellt ist. Diese Sichtweise auf die Anwendung ist wiederum eine andere als die Sicht eines Entwicklers, der den Quellcode programmiert und eine präzise Beschreibung der Klassen, Methoden und Datentypen verlangt.

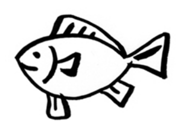
Das Besondere an der UML liegt darin, dass sie dieses weite Spektrum abdecken kann. Jedes einzelne Diagramm kann in einem festgelegten Detailgrad erstellt werden – für das Management, den Anwender oder für den Entwickler. Nicht jeder Beteiligte muss auch jedes Diagramm kennen. Es geht vielmehr darum, das komplexe Problem der Softwareentwicklung in seiner Gesamtheit nach und nach zu erfassen. Dies ist auch nur in einem iterativ-inkrementellen Prozess möglich.

Gerade bei dem Feststellen der Anforderungen an eine Anwendung gehen Sie zunächst von der Wolken- und/oder von der Meeresspiegel-Perspektive aus; je nachdem, mit welcher Personengruppe Sie kommunizieren. Um die Funktionen der zukünftigen Anwendung zu ermitteln, eignen sich Anwendungsfalldiagramme besonders gut, die im Sprachgebrauch auch meist als Use-Case-Diagramme bezeichnet werden. Für die Darstellung betrieblicher Abläufe eignen sich Aktivitätsdiagramme besonders gut. Auch diese können Sie bei Projektbeginn sowohl in Kooperation mit dem Management des Kunden für die Ermittlung von Zuständigkeiten und globalen betrieblichen Prozessen, als auch auf Anwenderebene für eine Detailbeschreibung einzelner Geschäftsprozesse verwenden.









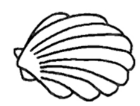


Abbildung 3.29: Verschiedene Perspektiven eines UML-Diagramms

Profitipp

Wechseln Sie innerhalb eines UML-Diagramms nicht die die Perspektive! Die gewählte Perspektive können Sie durch die Verwendung der Symbole in Abbildung 3.29 an dem jeweiligen Diagramm kennzeichnen.

Die benötigte Funktionalität: Anwendungsfälle

Nach den Erkenntnissen heutiger Softwareentwicklung wird in den ersten Schritten eines objektorientierten Projekts die benötigte Funktionalität von den Vertretern der Kundenseite ermittelt. Die UML bietet dazu die Anwendungsfalldiagramme an.

Abbildung 3.30 zeigt ein sehr globales Anwendungsfalldiagramm für die zu erstellende Seminarverwaltung aus der Wolkenperspektive. Nach Gesprächen mit dem Kunden hat sich herausgestellt, dass die zu erstellende Anwendung wesentlich mehr als nur Seminare verwalten soll, nämlich auch Dozenten, Kunden, Rechnungen usw.

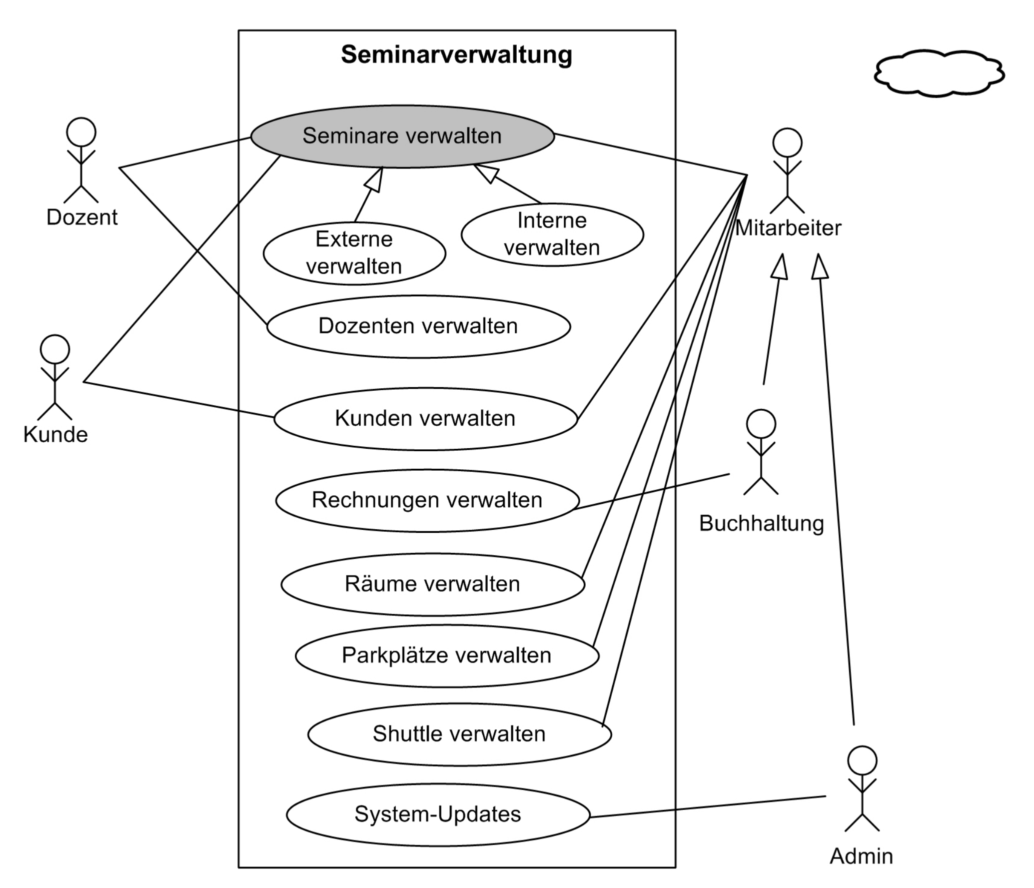


Abbildung 3.30: Anwendungsfall der Seminarverwaltung aus der Wolkenperspektive

Die Ellipsen beinhalten je eine globale Funktionalität. Die Bezeichnung beinhaltet zumeist ein Verb. Globale Perspektiven bestehen in der Regel aus *Verwaltungen*, die ihrerseits wiederum eine Vielzahl von Funktionen anbieten.

Neben der Ermittlung von Funktionen sollten Sie auch die wichtigsten Akteure identifizieren, die in jeden Anwendungsfall involviert sind. Akteure sind dabei meist Personen, Personengruppen oder Rollen, wie ein Sachbearbeiter, ein Kunde oder ein Administrator. Jede Rolle hat spezifische Berechtigungen in der Anwendung und wird meist über einen Anmeldevorgang am System identifiziert. Bei einem Akteur kann es sich jedoch auch um eine Institution handeln, beispielsweise um ein Kreditinstitut, bei dem ein Autohändler eine Schufa-Auskunft erbittet. Es kann ebenso eine Technologie sein, die selbst agiert. Wenn zum Beispiel eine Alarmanlage über das Mobilfunknetz eine SMS mit einer Einbruchsmeldung versendet, tritt die Alarmanlage gegenüber dem Mobilfunknetz als Akteur auf.

Sowohl bei den Anwendungsfällen als auch bei den Akteuren können Sie Vererbungspfeile verwenden. Die „Ist ein“-Phrase trifft auch hier zu. So ist die Verwaltung sowohl von externen als auch von internen Seminaren eine Seminarverwaltung, eben nur spezi-

eller. Ob sich aus diesen Vererbungen auch Klassenhierarchien aufbauen, ist zu diesem Zeitpunkt noch völlig unklar. Es geht nur darum, Funktionalität und deren Abhängigkeiten zu ermitteln. Ebenso existiert bei dieser fachlichen Modellierung noch keinerlei Bezug zu irgendeiner Programmiersprache. Sie könnten die Seminarverwaltung zu diesem Zeitpunkt auch in Java, ASP.NET oder C# realisieren. Die verwendete Technologie ist erst bei der technischen Modellierung im objektorientierten Design von Bedeutung.

Die Vererbung bei Akteuren deutet stets auf ein Rollensystem hin, bei dem einzelne Personengruppen spezielle Berechtigungen erwerben. So hat jeder Mitarbeiter Zugriff auf die Seminar-, Kunden-, Raum-, Parkplatz- und Shuttleverwaltung. Die Buchhalter sind spezielle Mitarbeiter, die auch die spätere Rechnungsverwaltung bedienen können. Diese Personen sind auch im aktuellen Geschäftsprozess die einzigen, die Rechnungen verfassen dürfen. Weil Buchhalter ja auch Mitarbeiter sind, können sie natürlich auch alle Dienste eines gewöhnlichen Mitarbeiters nutzen.

Ein Administrator ist eine einzige berechtigte Person, die in Zukunft Systemupdates in die Anwendung einspielen kann. Die Vererbungen der Akteure spiegeln also eine Klassenhierarchie der Benutzergruppen wider.

Neben der Vererbung existieren noch die <<*include*>>- und <<*extend*>>-Beziehungen zwischen Anwendungsfällen. Die Grenze zwischen einer <<*extend*>>-Beziehung und einer Vererbung ist fließend und liegt im Ermessen des Analytikers. Als Regel kann man aufstellen, dass die <<*extend*>>-Beziehung meist nur dann benutzt wird, wenn sich zwei Anwendungsfälle nur in genau einer Eigenschaft unterscheiden.

Beispiel

Was unterscheidet eine Bestellung von einer Eilbestellung? Eine Eilbestellung soll schneller am Ziel ankommen. Aber wie ist diese besondere Bestellung gekennzeichnet, was macht sie aus? Eine Eilbestellung besitzt im Gegensatz zu einer normalen Bestellung eine Priorität! Genau diese Priorität erweitert eine gewöhnliche Bestellung um das Merkmal des schnellen Versands.

Ob später daraus zwei separate Klassen erstellt werden oder lediglich ein Prioritäts-Flag zu einer gewöhnlichen Bestellung hinzugefügt wird, bleibt den Entwicklern überlassen. Die Aussage, dass eine Eilbestellung eine spezielle Bestellung ist, triff auf jeden Fall zu, sodass die Vererbung auch nicht falsch sein kann.

Abbildung 3.32 zeigt die Visualisierung der <<*extend*>>-Beziehung. Wenn Sie das Unterscheidungsmerkmal ermittelt haben, ist es sehr sinnvoll, dieses Merkmal auch im UML-Diagramm festzuhalten, damit diese Erkenntnis nicht im späteren Projektverlauf verloren geht.

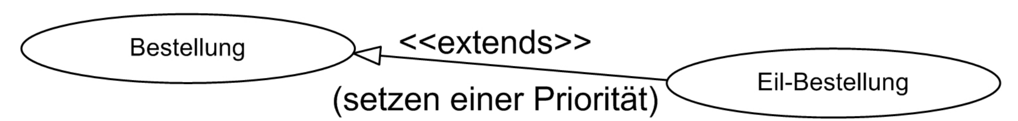


Abbildung 3.31: Extends-Beziehung der UML an einem Beispiel

Nachdem Abbildung 3.30 die gewünschten Funktionen der Anwendung auf Wolkenebene beschrieben hat, sollten Sie die Betrachtung noch etwas detaillierter durchführen. Dabei wird jeder einzelne Anwendungsfall der Wolkenperspektive genauer betrachtet und es wird jeweils ein neues Diagramm erstellt.

Profitipp

Vermeiden Sie es zu versuchen, jeweils ein globales Diagramm zu erstellen, das alle Funktionen enthält. Schon bei einer etwas komplexeren Anwendung werden Sie scheitern!

Abbildung 3.32 beschreibt die Funktion, Seminare zu verwalten, genauer. Man wechselt zur Drachenperspektive. Bei Bedarf kann man auch direkt den Meeresspiegel betrachten. Die Meeresspiegelperspektive enthält dann die Funktionen, die unmittelbar über die Anwendung erreichbar sind, beispielsweise über Schaltflächen in dem jeweiligen Verwaltungssystem.

Typischerweise werden Anwendungsfälle bis auf Meeresspiegelebene betrachtet. Diese Anwendungsfälle werden als Business-Use-Cases bezeichnet, die sich eher an den betrieblichen Geschäftsprozessen orientieren. Wirft man einen Blick unter das Wasser, so erstellt man System-Use-Cases, die interne Funktionen der Anwendung beschreiben, die ein Anwender nicht wahrnimmt und die nur für die Entwickler von Bedeutung sind. Um die Anzahl der erzeugten Diagramme nicht explodieren zu lassen, werden System-Use-Cases meist vermieden und spielen nur bei ganz speziellen Sachverhalten oder in hoch komplexen und unübersichtlichen Anwendungen eine Rolle.

Profitipp

Beachten Sie, dass in grafischen Anwendungsfällen keinerlei Reihenfolge der Funktionen existiert. Die Ellipsen von zusammengehörigen Funktionen werden jedoch oft nahe beieinander gezeichnet. Genauso werden ähnliche oder abgeleitete Akteure räumlich nah angeordnet. Üblicherweise werden externe Akteure auf der linken Seite des Systems und interne (Firmen-)Mitarbeiter auf der rechten Seite untergebracht.

Das Anlegen und Ändern von Seminaren wird stets in Absprache mit den Dozenten durchgeführt. Die Mitarbeiter suchen sich anhand der Raumplanung mögliche Termine für die Seminare aus, die dann dem Dozenten mitgeteilt werden. Hat ein Dozent Zeit, so wird er diesem Termin zugeordnet. Die <<*include*>>-Beziehung besagt, dass der inkludierte Anwendungsfall – hier das Zuordnen des Dozenten – nie allein ausgeführt wird. Der eingebundene Anwendungsfall ist also immer in einem größeren Kontext zu sehen. In diesem Fall ist dieser Kontext die Terminzuordnung. In der Implementierung wird ein solcher eingebundener Anwendungsfall als interne Hilfsmethode gehandhabt, die nicht direkt vom Benutzer angesprochen werden kann.

Sowohl der Kunde als auch der Mitarbeiter kann über eine Suchfunktion nach Seminaren suchen. Ein Kunde kann sich dann zu einem Seminartermin an- und auch wieder

abmelden. Wenn beispielsweise der Dozent erkrankt ist, kann ein Mitarbeiter ein Seminar stornieren. Dabei sollen Nachrichten an bereits angemeldete Kunden automatisch versendet werden. Zusätzlich kann ein Seminar durch einen Mitarbeiter für eine zukünftige statistische Auswertung archiviert werden.

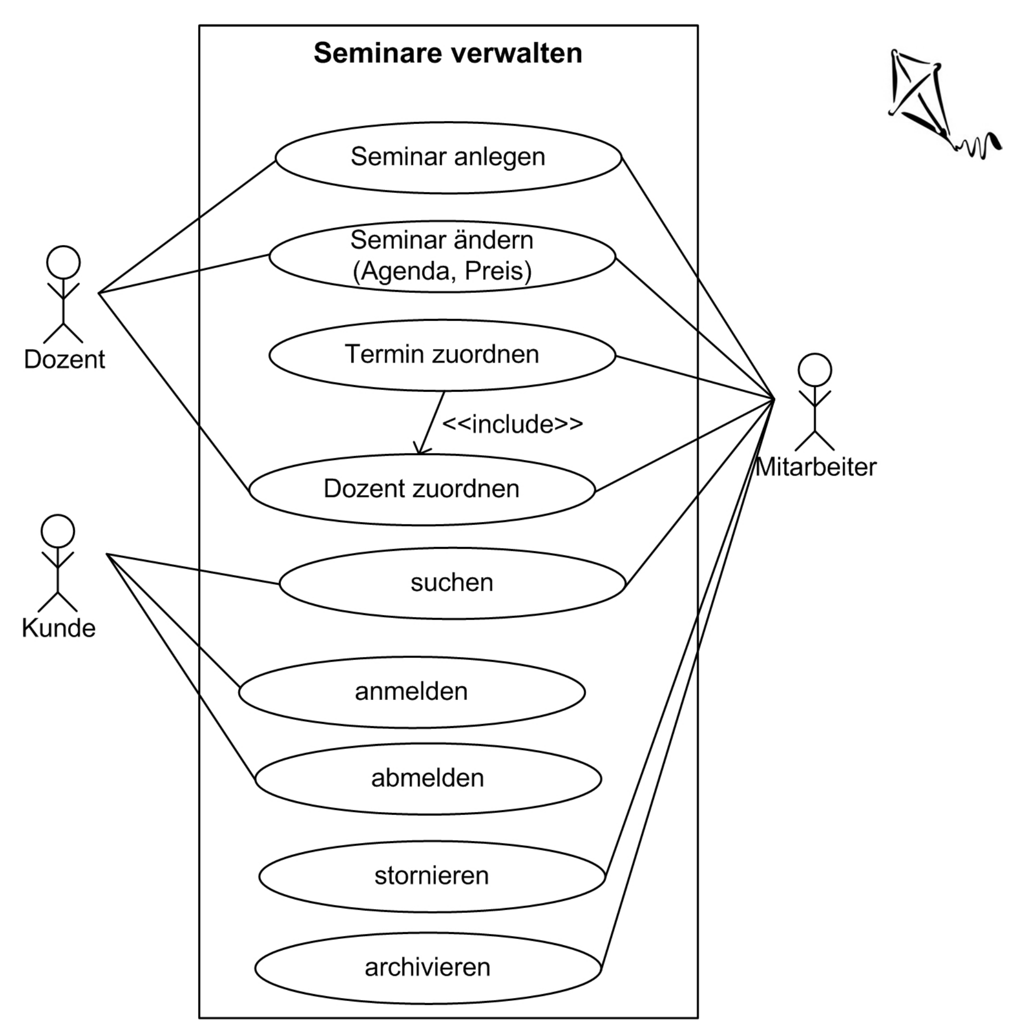


Abbildung 3.32: Anwendungsfall der Seminarverwaltung aus der Drachenperspektive

Ihnen ist vielleicht aufgefallen, dass die textuelle Beschreibung bereits ausreicht, um den Sachverhalt des Anwendungsfalldiagramms zu beschreiben. Wieso ist dann noch das Diagramm notwendig? Die Antwort liegt darin, dass das Diagramm lediglich die Diskussionsgrundlage liefert. Die UML ist also lediglich ein Hilfsmittel, um sich einer guten Spezifikation zu nähern. Die Diagramme und auch deren Iterationen und Weiterentwicklung während der Analyse dienen zwar der Dokumentation, sind aber nicht selbst das Ziel des Prozesses. Das Ziel ist es vielmehr, dass alle Projektbeteiligten eine gemein-

same Sprache finden und sich darüber einigen, welche Funktionen die zukünftige Software realisieren soll.

Profitipp

Wenn Sie UML-Diagramme über mehrere Iterationen entwickeln, überschreiben Sie bitte nicht die alten Versionen. Verwenden Sie besser eine Versionsverwaltung wie Subversion SVN. So können Sie im Nachhinein den Projektverlauf und auch Designentscheidungen besser nachvollziehen.

Auch wenn sich solche Aussagen trivial und fast selbstverständlich anhören, werden Sie in der fachlichen Modellierung bald feststellen, dass eine korrekte und präzise Beschreibung selbst einfacher Sachverhalte oft schwierig ist und zu weiche Aussagen oft von anderen Projektbeteiligten anders interpretiert werden.

Beispiel

Sie möchten von einem Softwareunternehmen eine Software entwickelt bekommen, die der Funktionalität von Microsoft Word 2003 entspricht. Erstellen Sie ein grafisches Anwendungsfalldiagramm aus der Wolken- und Meeresspiegelperspektive, ohne den Begriff „Microsoft Word 2003“ und „Textverarbeitung“ zu verwenden.

Bei dieser Übung wird als Lösung für die Wolkenebene oft Funktionalität beschrieben wie

- Texte bearbeiten
- Bilder einfügen
- Tabellen einfügen

und auf der Meeresspiegelebene

- Dokument laden/speichern/drucken
- fett/kursiv/unterstrichen
- Schriftgröße und -art ändern

Diese Funktionen sind zunächst nicht falsch. Wenn dies die Basis für eine objektorientierte Analyse darstellen soll, kann Ihnen das Softwareunternehmen auch Microsoft Excel oder PowerPoint liefern! Sie müssen also darauf achten, dass die Funktionen nicht so weich und allgemeingültig definiert werden, dass Sie zwar mit allen Beteiligten einen Konsens in der Analyse erreichen, später jedoch eine Enttäuschung bei der Vorstellung des Prototyps erleben.

Eine sprachliche Präzision der Anforderungen zu erreichen, ist nicht mal eben zwischendurch erledigt, sondern ein Prozess, der eine hohe Kompetenz und Konzentration von allen Beteiligten erfordert. Sie müssen also eine Beschreibung finden, die auf Microsoft Word zutrifft, aber auf keine Tabellenkalkulation oder Präsentationssoftware.

Ein Unterscheidungsmerkmal ist sicherlich die Bearbeitung von DIN-A4-Seiten zum Ausdruck, inklusive Formatierung der Seitenränder, Kopf- und Fußzeilen. Ein weiteres besonderes Merkmal einer Textverarbeitung ist die Verwaltung von Absätzen, Tabulatoren und Überschriften unter Verwendung von Formatvorlagen. Auch die Kontrolle von Rechtschreibung und Grammatik ist in einer Textverarbeitung sicherlich wichtiger als in anderen Anwendungen.

Der Informationsgehalt eines grafischen Use Cases ist, gerade bei komplexen Anwendungen, gering. Die Akteure und die gewünschten Funktionen könnten auch in eine Tabelle kompakter dargestellt werden.

Deshalb kann man in einem zweiten Schritt jeden Anwendungsfall nochmals genauer betrachten. Die textuellen Schablonen – die man beispielsweise als Vorlage in einer Textverarbeitung hinterlegen kann – sind zwar nicht in der UML standardisiert, werden jedoch häufig mit folgender Struktur befüllt:

- Name der gewünschten Funktion; identisch mit dem Namen aus dem grafischen Anwendungsfall.
- Globale Zielsetzung bei erfolgreicher Ausführung dieses Anwendungsfalls.
- Handelt es sich um einen primären, sekundären oder optionalen Anwendungsfall. Dies ist ein erster Anhaltspunkt für eine spätere Priorisierung.
- Erwarteter Zustand vor Beginn dieses Anwendungsfalls.
- Erwartetes Ergebnis nach erfolgreicher Ausführung.
- Erwarteter Zustand, falls Ziel nicht erreicht werden kann. Fehlschläge sind jedoch noch nicht technisch zu betrachten, z. B. wenn während einer Anmeldung das Netzwerk gestört ist. Vielmehr sind Fehlschläge innerhalb der Geschäftsprozesslogik gemeint, wie das Anmelden für ein bereits ausgebuchtes Seminar.
- Die beteiligten Rollen oder Personen können den grafischen Anwendungsfällen entnommen werden.
- Das auslösende Ereignis ist ein Trigger, bei dessen Auftreten der Anwendungsfall gestartet wird. Dieser Trigger muss nicht innerhalb des Systems auftreten; es kann beispielsweise der Anruf eines Kunden sein.
- Die Beschreibung gibt in kurzen nummerierten Stichpunkten wider, wie der kürzeste Weg zum Erfolg des Anwendungsfalls lautet. Dieser kürzeste Pfad lässt sich in der Regel bei Verwendung einer agilen Vorgehensweise sehr schnell in einem frühen Prototyp realisieren, der bereits für eine große Zahl an Fällen einsatztauglich ist.
- In einem separaten Punkt werden typische Erweiterungen des Funktionsumfangs im Gegensatz zur Beschreibung aufgeführt.
- Der letzte Punkt einer textuellen Schablone beinhaltet zumeist Alternativen zum kürzesten Weg, einen Anwendungsfall zu durchqueren. Wenn eine Alternative an eine Bedingung gekoppelt ist, sollten Sie diese unbedingt angeben.

Profitipp

Textuelle Anwendungsfallbeschreibungen bieten einen fließenden Übergang zwischen grafischen Anwendungsfällen und Aktivitätsdiagrammen, die sich vollständig auf Abläufe im Geschäftsprozess konzentrieren.

Abbildung 3.33 zeigt einen exemplarischen textuellen Anwendungsfall, der die Anmeldung an einem Seminar beschreibt. An den möglichen Fehlschlägen erkennen Sie sofort Fehlerklassen, die bei der Ausführung des Anwendungsfalls zum Einsatz kommen könnten. Diese Klassen werden im vierten Kapitel dieses Buches behandelt. In der Analyse müssen Sie sich mit Ihrem Auftraggeber darauf verständigen, was beim Auftreten dieser Fälle geschehen soll.

Bei der Beschreibung, Erweiterung und den Alternativen erkennen Sie erstmalig eine Abfolge in der UML. Bei den Alternativen werden stets die Bedingungen angegeben, bei denen die alternative Ausführung eintritt.

Seminar buchen
Ziel: Anmeldebestätigung an den Kunden geschickt. **Vorbedingung:** --- **Nachbedingung Erfolg:** Kunde ist angemeldet. **Nachbedingung Fehlschlag:** Mitteilung an Kunden, das Veranstaltung ausgebucht ist, ausfällt oder nicht existiert. **Akteure:** Kunde, Mitarbeiter **Auslösendes Ereignis:** Anmeldung des Kunden liegt vor. **Beschreibung:** 1. Kundendaten abrufen 2. Seminar prüfen 3. Anmeldebestätigung erstellen **Erweiterung:** 1a. Kundendaten aktualisieren 1b. Wenn Kunde MA einer Firma ist, Firmendaten erfassen bzw. wenn vorhanden, dann abrufen und aktualisieren 1c. Zahlungsmoral prüfen **Alternativen:** 1a. Neukunden erfassen, wenn Kunde noch nicht existiert 1b. Auf alternative Veranstaltungen hinweisen, wenn ausgebucht 1c. Mitteilung „Falsche Veranstaltung“, falls Veranstaltung nicht existiert und auch nichts ähnliches

Abbildung 3.33: Textuelle Anwendungsfallbeschreibung

Abläufe im Geschäftsprozess: Aktivitätsdiagramme

Aktivitätsdiagramme geben die Struktur eines Prozesses als Fluss dynamisch wider. Die ursprüngliche Notation, auf der die Aktivitätsdiagramme aufbauen, existiert bereits seit 1980 unter dem Namen der Programmablaufpläne (PAP). Diese Diagramme sind nach DIN 66001 genormt und werden noch heute zur Dokumentation von sequenziellen Anweisungen, Verzweigungen und Schleifen in einem Quellcode verwendet. Genau dies ist auch mit den Aktivitätsdiagrammen auf Fisch- und Muschelebene – also nah am Entwickler – möglich und auch üblich.

Eine größere Bedeutung innerhalb der UML haben die Aktivitätsdiagramme jedoch, um Geschäftsprozesse innerhalb des Unternehmens abzubilden, die im Anschluss daran ganz oder teilweise in einem technischen System abgebildet werden. Aber auch wenn keine Abbildung in einer (PHP-)Anwendung erfolgt, ist allein die Definition der Anwendungsfälle und der Geschäftsprozesse ein Mehrwert für Ihren Kunden.

Diese Abbildung kann auf zwei verschiedene Arten erfolgen, die auch vermischt werden können. Zum einen können Sie Datenabläufe modellierten, beispielsweise, wie das Aufnehmen einer Bestellung bis zu deren Versand in ihrem Unternehmen abläuft. Dies wird als Daten- oder Kontrollfluss bezeichnet und ist in Abbildung 3.34 oben rechts dargestellt.

Wenn Sie bereits Objekte identifiziert haben, können Sie auch die Weitergabe dieser Objekte innerhalb Ihres Unternehmens festhalten. So kann es sich bei der Aktion 1 in Abbildung 3.34 unten rechts um den Vorgang der Erfassung einer Bestellung handeln, die ein Bestellungsobjekt erzeugt und zurückliefert. Dieses Objekt wird dann in Aktion 2 vom Versand weiterverarbeitet.

Jede Aktion wird in einem abgerundeten Viereck dargestellt und sollte in ihrer Beschreibung eine Tätigkeit – also ein Verb – enthalten. Wenn Sie in einem Diagramm an eine Seitenbeschränkung gelangen, können Sie so genannte Wurmlöcher einfügen, wie es in der Abbildung mit dem Kreis A dargestellt ist. Um eine hohe Übersicht zu gewährleisten, sollten Sie jedoch nach Möglichkeit auf dieses Mittel verzichten. Die Wurmlöcher können jedoch auch auf ein anderes Aktivitätsdiagramm verweisen.



Abbildung 3.34: Aktionen, Wurmlöcher, Kontroll- und Objektflüsse

Ein Aktivitätsdiagramm beginnt stets mit einem schwarzen Startpunkt und endet mit einem weißen Punkt mit teilweiser schwarzer Füllung. In der UML2 ist noch ein irregulärer Endpunkt hinzugekommen, der dann erreicht wird, wenn der Geschäftsprozess nicht erfolgreich beendet werden kann (vgl. dazu auch das Feld *Nachbedingung Fehlschlag* in einem textuellen Anwendungsfall). Generell geben die textuellen Felder Beschreibung, Erweiterung und Alternativen eines Anwendungsfalldiagramms die gleichen

Informationen wieder wie ein Aktivitätsdiagramm. In einem Aktivitätsdiagramm sind sie lediglich grafisch aufbereitet. Auf Grund der hohen Redundanz ist zu überlegen, ob sowohl die textuellen Anwendungsfälle als auch die Aktivitätsdiagramme in einem Projekt vollständig ausgearbeitet werden.

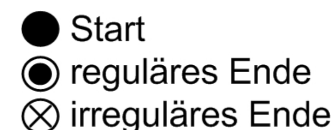


Abbildung 3.35: Anfang und Ende eines Aktivitätsdiagramms

Abbildung 3.36 zeigt die Darstellung einer Verzweigung, einer Verzweigung mit gleichzeitiger Zusammenfügung sowie eine reine Zusammenfügung hinter einer Verzweigung. Bei einer Verzweigung muss stets an allen Ausgängen der Raute eine Bedingung angegeben werden, bei deren Eintreffen dieser Pfad gewählt wird. Bedingungen in einem Aktivitätsdiagramm werden stets in eckigen Klammern angegeben. Bei den Programmablaufplänen wurden die Bedingungen noch in die Raute selbst geschrieben. Das hatte dazu geführt, dass die Rauten eine überdimensionale Größe eingenommen hatten.

[x<0] [x>0]
[x=0]
[x<0]
[x>=0]

Abbildung 3.36: Verschiedene Verzweigungen

Ebenfalls neu hinzugekommen in die UML2 sind die Symbole der Abbildung 3.37. Das linke Symbol erinnert an eine Sanduhr und bedeutet den Ablauf einer absoluten oder relativen Zeitspanne, die in der Regel als Beschriftung unter dem Symbol angegeben wird. So kann das Symbol beispielsweise nach einer Rechnungsstellung mit einer Zeitdauer von zwei Wochen im Aktivitätsdiagramm platziert werden. Im Anschluss daran erfolgt mittels einer Verzweigung eine Prüfung, ob die Rechnung bereits bezahlt ist. Ist dies nicht der Fall, wird die erste Mahnung versendet.

Die beiden anderen Symbole wurden aus der Specification and Description Language (SDL) übernommen und beschreiben den Empfang einer Nachricht bzw. deren Absendung. Dabei kann es sich um eine Nachricht im weitesten Sinne halten, von einem digitalen Signal bis hin zu einem eingehenden Telefonat, einer E-Mail oder einer Faxnachricht. Der Empfang einer E-Mail kann beispielsweise als Trigger bzw. als auslösendes Ereignis eines Anwendungsfalls verwendet werden, vgl. dazu Abbildung 3.33. Generell dienen Aktivitätsdiagramme dazu, die Abläufe innerhalb eines Anwendungsfalls darzustellen.



Abbildung 3.37: Zeitereignis, empfangene und gesendete Nachricht

Wenn Sie bereits Objekte aus dem Geschäftsprozess erkennen, können Sie diese im Aktivitätsdiagramm auch direkt benennen. Dies ist in Abbildung 3.38 dargestellt. Achten Sie darauf, dass das Viereck hier nicht abgerundet ist. Die eckigen Klammern in diesem Viereck stellen keine Bedingung dar, sondern einen Zustand, in den das Objekt zu diesem Zeitpunkt versetzt wird. Dieser Zustand spielt wieder im Zustandsdiagramm im vorletzten Teil dieses Kapitels eine Rolle. So kann das Objekt *Bestellung* beispielsweise in die Zustände *in Bearbeitung*, *ausgeliefert*, *bezahlt* und *storniert* versetzt werden oder das Objekt *Seminar* in die Zustände *existiert*, *buchend*, *laufend*, *abgesagt* und *durchgeführt*. Bereits im Aktivitätsdiagramm sollten Sie sich – wenn Sie Objekte entdecken – über mögliche Zustände erste Gedanken machen.



Abbildung 3.38: Ein Objekt wird in einen Zustand versetzt

Gegenüber den Programmablaufplänen haben die Aktivitätsdiagramme eine weitere Erweiterung erhalten. Mit der in Abbildung 3.39 abgebildeten Syntax können Sie parallele Abläufe darstellen. Aus Sicht des Geschäftsprozesses (Wolken- bis Meeresspiegelserspektive) sind dies parallel ablaufende Vorgänge im Unternehmen.

Beispiel

Wenn beispielsweise eine Anfrage für ein Angebot eingeht, prüft der Einkauf die Preise für das Material, während der Vorstand den ersten Kontakt zum potenziellen Kunden aufnimmt.

Parallele Abläufe können Sie jedoch auch technisch betrachten. In der Fisch- und Muschelperspektive kann beispielsweise ein Dokument gedruckt werden, während der Anwender gleichzeitig weitere Funktionen der Anwendung nutzen kann. In diesem Fall bedeutet dies die Anwendung von Multi-Threading oder Multi-Processing.

Das linke Symbol zeigt den Beginn eines parallelen Ablaufs, das nächste Symbol eine Synchronisation. Dabei müssen beide ankommenden Pfade beendet sein, damit die Verarbeitung fortgesetzt wird. Das dritte Symbol beinhaltet eine Synchronisation mit gleichzeitigem Start von zwei neuen Prozessen, wenn beide vorherigen beendet sind.

Sie können sogar einen beliebigen booleschen Ausdruck verwenden, der erfüllt sein muss, damit die Verarbeitung fortgesetzt wird. Das rechte Symbol setzt die Bearbeitung dann fort, wenn auf jeden Fall der ankommende Prozess A beendet ist und zusätzlich dazu der Prozess B oder C. Aus Sicht des Geschäftsprozess kann dies heißen, das auf

jeden Fall der Vorstand einer Entscheidung zustimmen muss und zusätzlich der Einkauf oder die Personalabteilung.

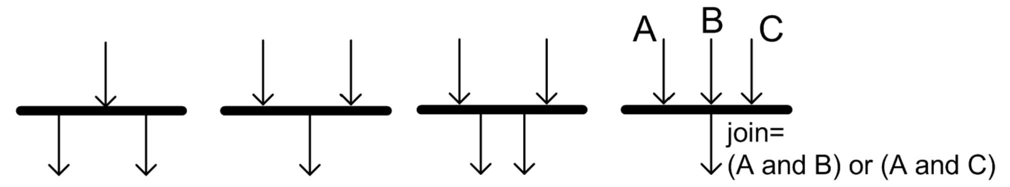


Abbildung 3.39: Teilung und Synchronisation im Aktivitätsdiagramm

Mit der Hilfe von Schwimmbahnen (Swim Lanes) können Sie den Fluss über Grenzen von Zuständigkeiten modellieren. Oft wird die Kommunikation zwischen Akteuren und dem System über Schwimmbahnen verdeutlicht. Das Aktivitätsdiagramm verläuft über die Schwimmbahnen. Die Aktionen, für die ein Akteur verantwortlich ist, werden dann in seinem Bereich gezeichnet.

Profitipp

Bitte verwenden Sie nicht zu viele Schwimmbahnen in einem Aktivitätsdiagramm! Mehr als vier Zuständigkeiten machen das Diagramm unübersichtlich. Versuchen Sie in diesem Fall, die Aktionen über mehrere Diagramme zu verteilen.

Zuständigkeit 1	Zuständigkeit 2	Zuständigkeit 3

Abbildung 3.40: Abbildung von Zuständigkeiten durch die Verwendung von Schwimmbahnen

Eine weitere Neuerung in der UML2 besteht darin, dass Sie in einem Aktivitätsdiagramm zwischen einer regulären Abarbeitung eines Geschäftsprozesses und einer Abarbeitung, die zu einem Fehlerfall führt, unterscheiden können. Diese Unterscheidung wurde erstmals in der textuellen Anwendungsfallschablone beschrieben.

Abbildung 3.41 zeigt die Aktion, bei der sich ein Kunde an einem Seminar anmelden will. Wenn dies erfolgreich ist, existiert ein neues Anmeldungsobjekt, das dann in der Datenbank abgelegt wird. Außerdem erhält der Kunde in diesem Fall automatisch eine Anmeldebestätigung.

Ist das Seminar aber bereits ausgebucht, schlägt der Versuch einer Anmeldung fehl. Das Ergebnis ist ein Fehlerobjekt, das durch das Dreieck gekennzeichnet wird. Dieser Fehler ist nicht so kritisch, dass deshalb die ganze (PHP-)Anwendung beendet werden muss. Stattdessen könnten dem Kunden alternative Termine angeboten werden. Sie können sich also im Aktivitätsdiagramm überlegen, wie Sie mit einem solchen Fehler umgehen wollen.

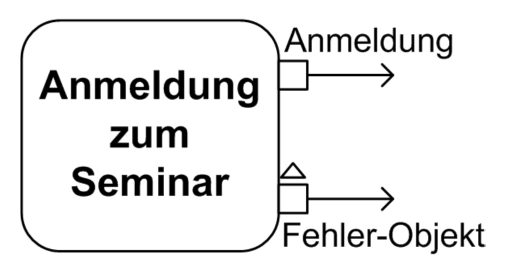


Abbildung 3.41: Reguläre und fehlerhafte Abarbeitung

Abbildung 3.39 zeigte bereits die parallele Abarbeitung von zwei oder mehreren Vorgängen. Zusätzlich existiert in der Notation der Aktivitätsdiagramme seit UML2 die Möglichkeit, eine Verarbeitung von Objektmengen zu beschreiben. Das linke Symbol der Abbildung 3.42 beschreibt als Aktion eine Mengenverarbeitung, die eine Eingangs- und eine Ausgangsmenge von Objekten behandelt.

Diese Verarbeitung lässt sich durch Unteraktivitäten genauer beschreiben. In dem rechten Teil der Abbildung ist zu erkennen, dass mit jeweils einem Objekt der Menge eine Aktion durchgeführt wird. Zusätzlich können Sie die Art der Verarbeitung angeben. In den Abbildungen können die Objekte der Eingangsmenge parallel verarbeitet werden. Die Objekte können also voneinander unabhängig abgearbeitet werden. Alternativ dazu wäre auch eine iterative Verarbeitung nach einem FIFO-Prinzip (first in, first out) denkbar.

Sie können jetzt vielleicht vermuten, dass es sich bei der Mengenverarbeitung um einen sehr seltenen Spezialfall handelt. Selbstverständlich sollten Sie es vermeiden, in jeden Vorgang eine Mengenverarbeitung hineinzuinterpretieren. Andererseits ist das Erkennen einer solchen Verarbeitung ein Mehrwert für die Beschreibung des Geschäftsprozesses.

Beispiel

Wenn Sie sich an Ihrer (PHP-)Anwendung morgens anmelden, sehen Sie in einem Unterfenster eine Liste von Aufträgen, die Sie zu erledigen haben. Die Liste ist nach Deadlines geordnet. Sie wählen die nächste wichtige Aufgabe aus und erhalten vom System Detailinformationen. Nun bearbeiten Sie diesen Auftrag und markieren ihn nach der Erledigung als fertiggestellt. Eine solche iterative Mengenverarbeitung ist beispielsweise in Microsoft Outlook im Modul der Aufgabenverwaltung integriert.

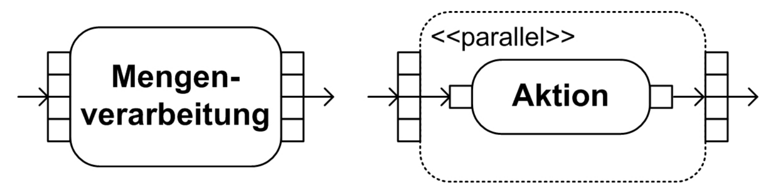


Abbildung 3.42: Abarbeitung einer Objektmenge

Die Notation der Aktivitätsdiagramme wurde nun ausreichend beschrieben. Als nächstes ist zu überlegen, wie Sie auf eine sinnvolle Art Aktivitätsdiagramme erstellen. Aktivitäts-Diagramme werden typischerweise aufgestellt, wenn einige zusammenhängende Anwendungsfälle fertig gestellt wurden. Als nächstes müssen Sie die grundsätzlichen Abläufe, die zur Erfüllung der Funktionalität der Anwendungsfälle notwendig sind, ermitteln. Dies ist am Besten über die Erstellung von Szenarien möglich.

Ein Szenario ist eine spezifische Sequenz von Aktionen, die das Verhalten des Systems unter bestimmten Bedingungen beschreibt. Dies ist beispielsweise

- ein Login
- ein Bestellvorgang
- eine Rechnungsstellung
- der Vorgang zur Auszahlung von Geld
- eine Angebotserstellung

Ein Szenario beschreibt also genau einen Workflow in einem Aktivitätsdiagramm. Bei der Analyse sollten Sie mit der Zeit, also über mehrere Iterationen, alle wichtigen Szenarien in Aktivitätsdiagrammen festhalten, da Sie ansonsten ganze Abläufe in der zur erstellenden Anwendung übersehen. Dadurch würden Sie auch ganze Klassen oder eine Vielzahl von Methoden nicht implementieren. Abbildung 3.43 zeigt ein erstes Beispiel dazu unter Verwendung von Schwimmbahnen.

Zunächst fällt auf, dass weder der Anfangspunkt, noch der Endpunkt des Aktivitätsdiagramms dargestellt ist. Außerdem existiert zwar ein Synchronisationspunkt, doch wo die Aufteilung der Prozesse beginnt, ist auf dem ersten Blick nicht ersichtlich.

Oft werden Aktivitätsdiagramme im ersten Schritt lediglich skizziert, um Gedanken festzuhalten und zu dokumentieren. Dabei werden Sachverhalte als selbstverständlich hingenommen. Wenn sich alle Beteiligten darüber einig sind, ist dies auch in Ordnung. Meist ist dies jedoch über den gesamten Prozess der Entwicklung nicht der Fall und führt später zu Missverständnissen. Sie sollten also jede Skizze auf jeden Fall präzisieren.

In der Realität steckt der Kunde seine Bankkarte in den Automaten und gibt die PIN ein. Noch während die Gültigkeit der Karte und der PIN vom Geldautomaten über eine Netzwerkverbindung zum Bankserver geprüft werden, kann der Kunde bereits den gewünschten Betrag wählen. Meist benötigt der Kunde genauso lange zum Überlegen wie der Geldautomat für die Kommunikation mit dem Server.

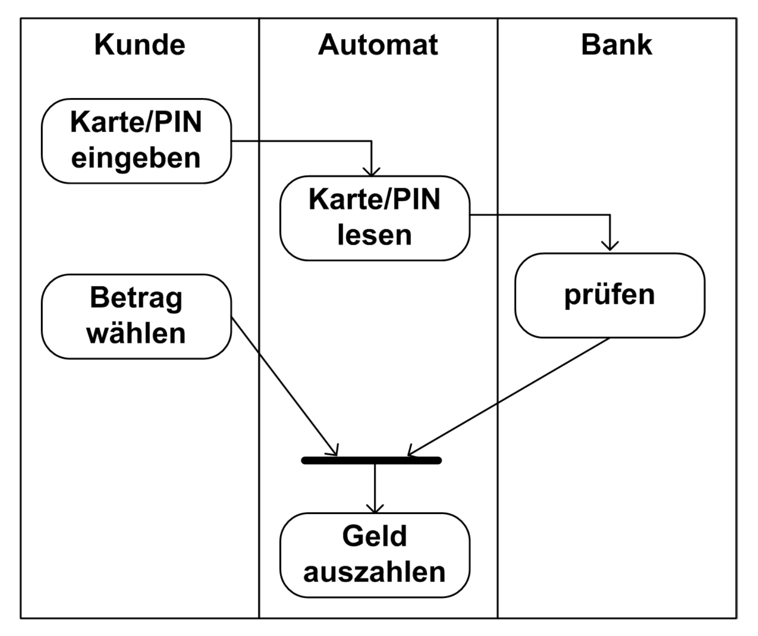


Abbildung 3.43: Geld abheben an einem Bankautomaten

Wenn die Prüfung erfolgreich ist und der Kunde den gewünschten Betrag zur Auszahlung gewählt hat, bekommt er das Geld ausgezahlt.

- Was jedoch geschieht, wenn die Karte und/oder die PIN ungültig sind?
- Was geschieht, wenn das Tageslimit des Kunden überschritten wird?
- Was geschieht bei der Eingabe eines ungültigen Betrags?
- Wie wird ein Netzwerkfehler zwischen dem Geldautomaten und dem Bankserver gehandhabt?

Die Lösung besteht darin, dass diese Szenarien hier nicht betrachtet werden. Abbildung 3.43 zeigt nämlich lediglich das Primärszenario bei einem erfolgreichen Durchlauf des Anwendungsfalls *Geld abheben*.

Profitipp

Sie sollten stets zuerst das Primärszenario ohne Ausnahmen modellieren. Dies entspricht der Beschreibung des textuellen Anwendungsfalls.

Versuchen Sie also nicht, alle Ausnahmen nachträglich in das Primärszenario einzufügen. Das Diagramm würde dadurch extrem unübersichtlich und man würde den einfachen Fall gar nicht mehr erkennen. In der oberen Auflistung wurden vier Sonderfälle definiert, die nicht zu einer regulären Abarbeitung des Anwendungsfalls führen.

Erstellen Sie also zu diesen vier Fällen eigene Aktivitätsdiagramme. Es ist kein Mehraufwand im Vergleich zur Integration aller Fälle in einem einzigen Diagramm!

Abbildung 3.44 zeigt zur Abwechslung ein Aktivitätsdiagramm auf Muschelebene, das eine Schleife in einem Quellcode beschreibt. In diesem Fall wird eine Variable *x* solange inkrementiert, wie sie kleiner als der Wert 10 ist.

Daran können Sie die vielfältige Anwendung dieses Diagrammtyps erkennen, der von grob beschriebenen Geschäftsprozessen bis zu sequenziellen Anweisungen, Verzweigungen und Schleifen im Quellcode reicht.

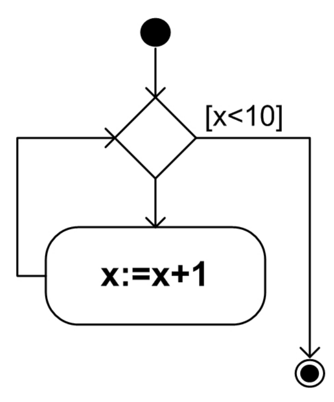


Abbildung 3.44: Ein Aktivitätsdiagramm zur Quellcodebeschreibung aus der Muschelebene

Beispiel

Entwerfen Sie ein Aktivitätsdiagramm auf Muschelebene, das für eine gegebene natürliche Zahl $n>0$ die Summe s und das Produkt p aller Zahlen von 1 bis n berechnet. So gilt beispielsweise für $n=5$ das Ergebnis $s=1+2+3+4+5=15$ und $p=1*2*3*4*5=720$. Das Verfahren besteht darin, in einer Schleife jede Zahl i von 1 bis n zu durchlaufen und in jeder Iteration i zu s zu addieren bzw. mit p zu multiplizieren.

Abbildung 3.45 zeigt die Lösung für dieses Beispiel. Als Übung können Sie dieses Aktivitätsdiagramm bereits mit den beschriebenen Möglichkeiten des zweiten Kapitels umsetzen. Für die Eingabe von *n* können Sie ein HTML-Formular mit einem Textfeld verwenden, dessen Daten dann zu einer PHP-Datei weitergeleitet werden. Dort lesen Sie den übergebenen Parameter aus und kodieren das Diagramm unter Verwendung einer *do-while*-Schleife und einer Verzweigung, die den Wert des Eingabeparameters prüft.

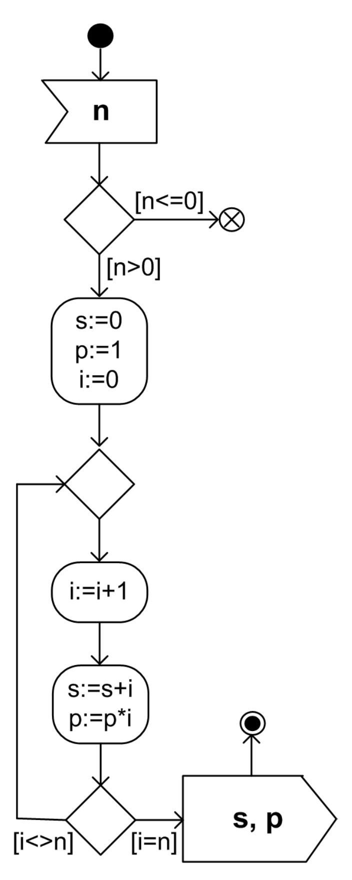


Abbildung 3.45: Lösung des Beispiels auf Muschelebene

Das folgende Beispiel zeigt ein Aktivitätsdiagramm auf Meeresspiegelebene, das also die Aktionen beschreibt, die ein Anwender an dem System durchführt. Die Interaktion wird durch die Verwendung von zwei Schwimmbahnen verdeutlicht.

In der Suchmaschine des Seminaranbieters gibt der Kunde den Begriff *PHP* ein. Zurückgegeben wird von dem Webserver eine Liste aller Seminare mit deren zukünftigen Terminen. Von diesen Terminen sucht sich der Kunde einen aus und gibt in der nächsten Eingabemaske seine persönlichen Daten, wie Name und Anschrift, ein. Sind alle Daten eingegeben, wird ein neues Buchungsobjekt erstellt und zum Server der Seminarverwaltung gesendet.

In diesem Szenario war noch genau ein Platz zu diesem Seminartermin frei, dieses Seminar war im Zustand *buchend*. Der Server nimmt die neue Buchung entgegen und prüft die vom Kunden eingegeben Daten auf Gültigkeit. Da alles in Ordnung war, wird die Buchung dem Seminartermin hinzugefügt. Danach ist die maximale Teilnehmerzahl für diesen Termin erreicht, sodass der Termin ausgebucht ist. Der Server der Seminarverwaltung sendet abschließend eine positive Buchungsbestätigung zum Kunden. Dies könnte natürlich auch über die Senden- und Empfangssymbole aus Abbildung 3.37 erfolgen.

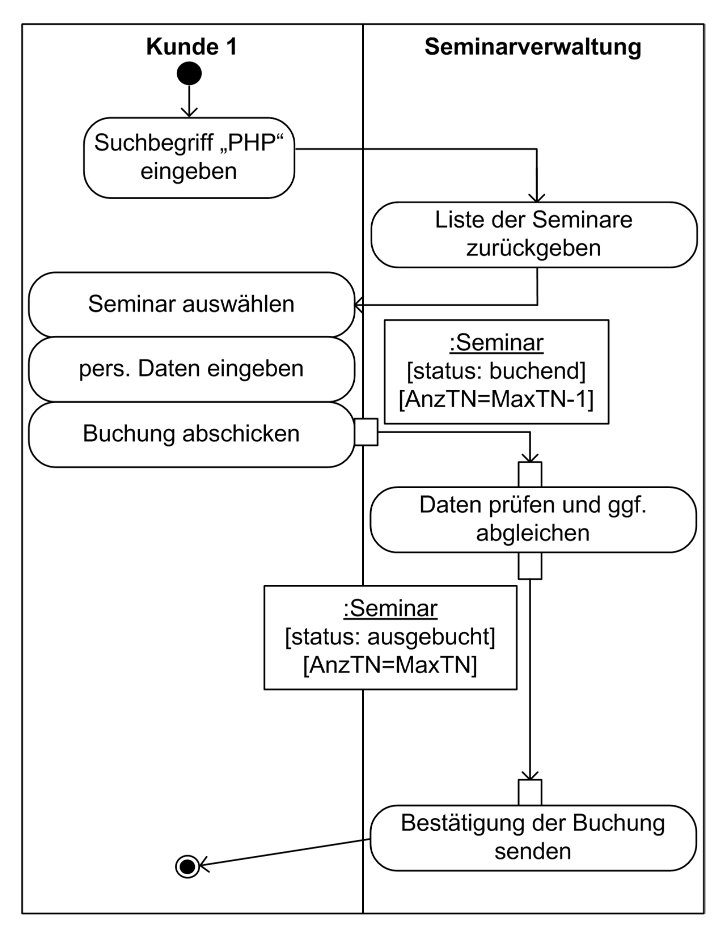


Abbildung 3.46: Erfolgreiches Buchen eines Seminars auf Meeresspiegelebene

Diesem positiven Beispiel wird in Abbildung 3.47 ein Fehlschlag gegenübergestellt. In diesem Szenario hat sich der Kunde den Link auf einen Seminartermin in den Bookmarks seines Internetbrowsers gemerkt und ruft diesen Link nun ab.

Er gibt auch hier seine persönlichen Daten ein und erzeugt ein Buchungsobjekt, das an den Server gesendet wird. Der Seminartermin ist jedoch bereits ausgebucht; der Kunde hat sich also zu spät angemeldet.

In diesem Fall ist vorgesehen, dass der Kunde eine Mitteilung erhält, dass dieser Termin bereits ausgebucht ist. Zusätzlich erhält er eine Liste mit anderen möglichen Terminen sowie eine Kontaktanschrift des Seminaranbieters. Das Ziel des Geschäftsprozesses besteht darin, den buchungswilligen potenziellen Kunden nicht zu verlieren.

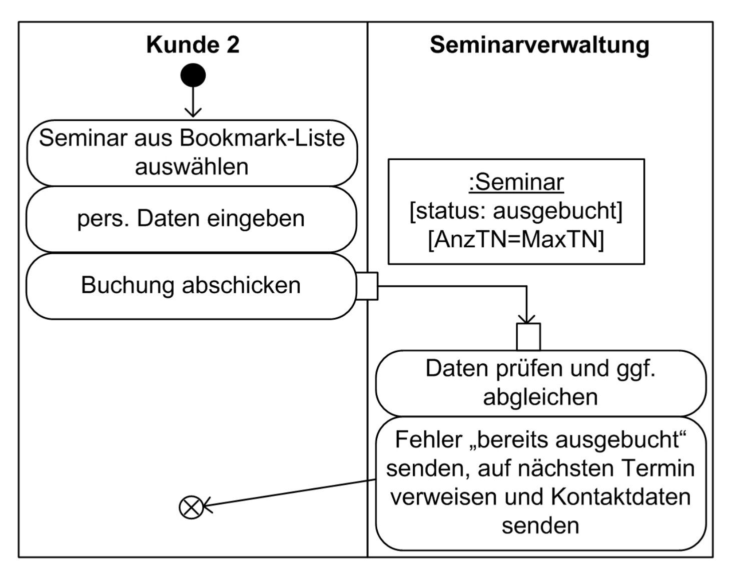


Abbildung 3.47: Buchungsversuch auf ein bereits ausgebuchtes Seminar

Meinung

Eine gute Übung ist es, wenn Sie im Internet nach bereits fertigen Aktivitätsdiagrammen suchen und diese kritisch beurteilen. Verstehen Sie den Ablauf? Ist er eindeutig spezifiziert? Was könnte man besser machen? Wenn Sie eine Vielzahl von Diagrammen gesehen und am besten mit anderen Personen diskutiert haben, werden Sie zu einem besseren Analytiker für objektorientierte Anwendungen.

Objekte und Klassen in der Analyse

In der Praxis begehen Entwickler oft den Fehler, Klassendiagramme zu früh zu erstellen, da sie ja möglichst bald entwickeln wollen und die Sachverhalte bereits sehr klar erscheinen. Dies ist jedoch eine trügerische Annahme.

In der UML existiert ein weiterer Diagrammtyp, der zu wenig Beachtung findet. Die Objektdiagramme sind an die Notation der Klassendiagramme angelehnt. Da ein Objekt eine Instanz, also ein Exemplar oder ein Beispiel einer Klasse darstellt, ist es sinnvoll, zunächst diese Beispiele zu betrachten, bevor man die abstrakteren Klassen modelliert; siehe dazu auch Abbildung 3.12, in der die Realität über die Objekte zu den Klassen abstrahiert wird.

Beispiel

Ein Kunde hat Ihnen das Beispiel aus Abbildung 3.48 aufgezeichnet, welches Sie im nächsten Projekt in einer PHP-Anwendung realisieren sollen. Der Kunde möchte gern ein Tool programmiert bekommen, mit dem man beliebig viele Punkte und Dreiecke in ein bereits bestehendes Koordinatensystem eintragen kann.

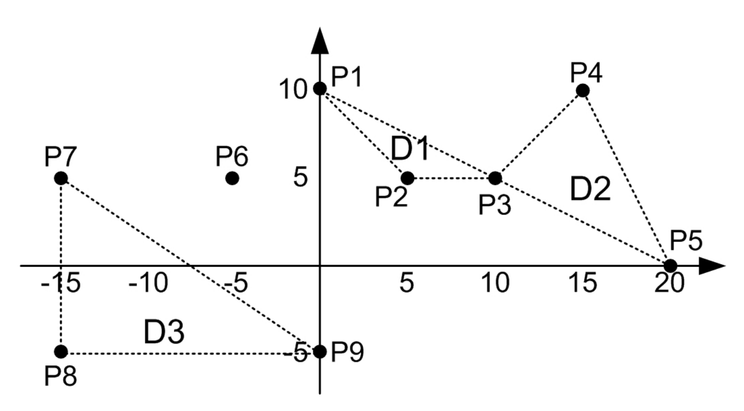


Abbildung 3.48: Von einem Kunden erstelltes Beispiel

Bevor Sie nun über dieses Beispiel diskutieren, sollten Sie es in ein Objektdiagramm überführen. Diese Umwandlung ist relativ einfach; viele Entwickler sehen sie deshalb als zu trivial an. Der Vorteil der Objektdiagramme besteht jedoch darin, dass Sie jedes beliebige Beispiel auf diese Weise in eine einheitliche Notation überführen können - genau dies ist der Sinn der UML. Zusätzlich sind Objektdiagramme so nah am Beispiel orientiert, dass der Kunde die Objektdiagramme selbst noch lesen und dadurch wie auch Anwender und andere Stakeholder einen Einstieg in die Modellierung finden kann.

Was also erkennen Sie in Abbildung 3.48? Es sind 3 Dreiecke und 9 Punkte zu erkennen. Es gibt also Dreiecke und Punkte. Damit haben Sie bereits die Klassen identifiziert.

Wie stehen die Klassen in Verbindung zueinander? Ein Punkt kann alleine existieren, siehe P6. Ein Dreieck kennt immer genau drei Punkte. Ein Punkt kann aber auch mehrere Dreiecke kennen, siehe P3.

Aus was bestehen ein Dreieck und ein Punkt? Ein Dreieck besteht aus drei Punkten. Ein Punkt wiederum besteht aus genau zwei Koordinaten. Alle diese Sachverhalte können Sie in der normierten Darstellung auch in Abbildung 3.49 erkennen.

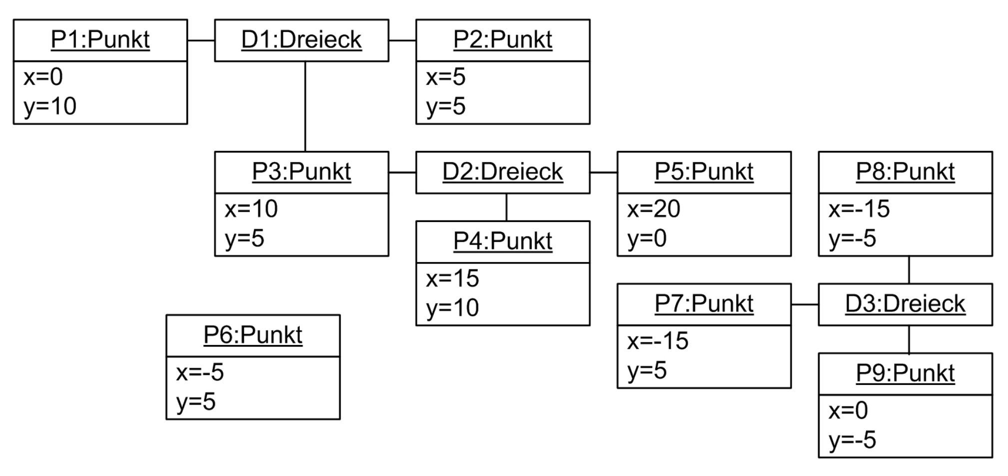


Abbildung 3.49: Objektdiagramm des Beispiels

Zentriert im oberen Viereck eines Objekts steht zunächst dessen Name und durch einen Doppelpunkt getrennt der Name der Klasse. Dies alles ist unterstrichen, um eine bessere Unterscheidbarkeit zu den Klassendiagrammen zu gewährleisten.

Besitzt ein Objekt zusätzliche Eigenschaften, werden sie in einem zweiten Viereck unter dem Namen benannt und deren aktuelle Wertausprägungen für das jeweilige Objekt eingetragen. Bei den Punkten sind dies die jeweiligen x- und y-Koordinaten. Wenn ein Objekt ein anderes Objekt kennt, wird diese Assoziation in einer Programmiersprache auch über eine Eigenschaft abgebildet. Die Kenntnis von Objekten untereinander wird jedoch in den Objekt- und auch in den Klassendiagrammen durch eine Linie gekennzeichnet. Der Fokus der Assoziation liegt darin, dass zwei Klassen miteinander kommunizieren, die aber ansonsten eigenständig sind.

Methoden werden in einem Objektdiagramm nicht berücksichtigt, da alle Objekte einer Klasse stets dieselben Methoden besitzen. Bei den Objektdiagrammen nähert man sich also der Modellierung über die gemeinsamen Eigenschaften von Objekten an.

Abbildung 3.50: Objekt ohne Klassenbeschreibung und anonymes Objekt

Abbildung 3.50 zeigt in der linken Darstellung, dass Sie den Namen der Klasse nicht bereits kennen müssen, wenn Sie ein Objektdiagramm zeichnen. *Frank* kann ein Kunde, Lieferant, Mitarbeiter oder nur irgendeine Person sein. Genauso können Sie auch anonyme Objekte erstellen, wenn Sie die Objekte nicht konkret benennen möchten. So definiert die rechte Darstellung der Abbildung „irgendeinen Kunden".

Für die Implementierung können Sie aus dem Objektdiagramm als Diskussionsgrundlage jedoch noch viel mehr erkennen. Zunächst können Punkte auch ohne Dreiecke existieren. Sie können also beliebige Punkte zeichnen. Wenn Sie aber ein neues Dreieck zeichnen wollen, müssen Sie bereits in dessen Konstruktor drei Punkte übergeben, da ansonsten das Dreieck nicht existieren kann.

Die nächste Frage, die Sie sich als aufmerksamer Entwickler stellen müssen, lautet: Können Sie aus drei beliebigen Punkten ein Dreieck erstellen? Dies geht sicherlich nicht, wenn die drei Punkte auf derselben x- oder y-Koordinate liegen. Denn dann würden Sie kein gültiges Dreieck, sondern eine Strecke erzeugen, die keinen Flächeninhalt besitzt. Außerdem könnten Sie keine Winkelberechnungen durchführen, die ja vielleicht als Methoden eines Dreiecks sinnvoll sind. Genügt es also zu prüfen, ob sich die drei x- und y-Koordinaten der Punkte unterscheiden? Dazu kann man ein Gegenbeispiel erzeugen:

P1x=1, P1y=1

P2x=2, P2y=2

P3x=4, P3y=4

Auch hier wird eine Strecke und kein Dreieck erzeugt. Im Konstruktor eines Dreiecks müssen Sie mit zwei der drei Punkten über die Geradengleichung $y=ax+b$ eine Gerade erzeugen und prüfen, ob der dritte Punkt auf dieser Geraden liegt. Wenn dies der Fall ist, darf das Dreieck nicht erzeugt werden.

Die Existenz des Dreiecks ist also von drei Punkten abhängig und jedes Dreieck kennt seine drei Punkte. Wenn ein Punkt verschoben wird, verändern sich auch alle Dreiecke, die aufgrund dieses Punktes existieren. Auch hier müssen Sie darauf achten, dass stets alle Dreiecke gültig bleiben.

Als Nächstes müssen Sie sich fragen, ob ein Punkt auch seine Dreiecke kennen muss. Auf den ersten Blick scheint dies nicht nötig zu sein. Was jedoch geschieht, wenn Sie einen Punkt löschen? In diesem Fall muss dieser zu löschende Punkt auch allen angeschlossenen Dreiecken den Befehl geben, sich zu löschen, da diese Dreiecke nicht ohne diesen Punkt existieren können. Jeder Punkt muss also auch eine Liste mit angeschlossenen Dreiecken verwalten. Und wenn ein Dreieck gelöscht wird, muss es allen drei Punkten ebenso mitteilen, dass es nicht mehr existiert. Denn ansonsten würden diese Punkte noch Referenzen auf ein Dreieck besitzen, das nicht mehr existiert.

Meinung

Dieser Ausblick, der von einem Objektdiagramm ausgeht und bis tief in die Implementierung reicht, zeigt die notwendige Disziplin und Sorgfalt, die man als Analytiker und auch als Entwickler bei der (objektorientierten) Programmierung besitzen muss. Dies wird noch unterstützt davon, dass Sie als Autor der Klassen Punkt und Dreieck bei Fehlern wesentlich leichter zur Verantwortung gezogen werden können als bei einem prozedural entwickelten Projekt, bei dem die Aufgaben mehrerer Entwickler nicht so stark voneinander trennbar sind.

Aus dem in Abbildung 3.48 dargestellten Objektdiagramm der Analyse müssen Sie im nächsten Schritt ein Klassendiagramm der Analyse erstellen. Der Fokus der Analyse besteht darin, die Klassen und deren Beziehungen zu ermitteln. Es wurde bereits festgestellt, dass die Klassen *Punkt* und *Dreieck* existieren, die sich gegenseitig kennen.

Die möglichen Darstellungen einer Assoziation finden Sie in Abbildung 3.51. Wenn keine Pfeile eingezeichnet sind, machen Sie noch keine Aussage darüber, welche Klasse welche andere kennt. Gerade zu Beginn der Analysephase ist dies üblich.

Zusätzlich wird in der Abbildung ausgesagt, dass K3-Objekte K4-Objekte kennen können. Ob K4-Objekte auch K3-Objekte kennen können, ist nicht spezifiziert worden. Wenn Sie eine Kenntnis explizit verbieten wollen, müssen Sie dies durch ein Kreuz darstellen. So ist es zwar sinnvoll, dass ein Richter Zugriff auf die Informationen eines Sträflings erhält, jedoch sollte dieser verständlicherweise aus Datenschutzgründen nicht den Wohnort und Familienverhältnisse des Richters kennen dürfen. Der untere Teil der Abbildung zeigt die gegenseitige Bekanntschaft zwischen Punkten und Dreiecken. Welche Objekte welche anderen Objekte kennen können, wird als Navigierbarkeit bezeichnet.

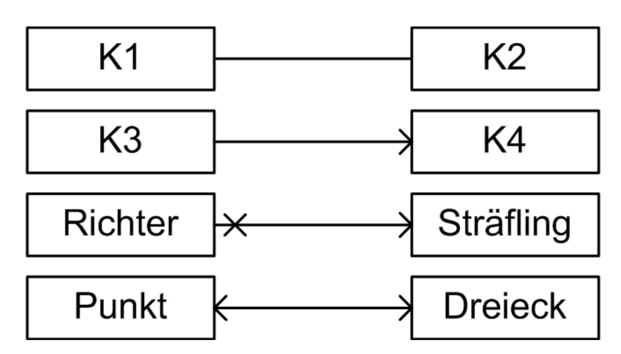


Abbildung 3.51: Navigierbarkeit der Assoziationen

Ein erstes Klassendiagramm der Analyse kann daher so aussehen, wie es in Abbildung 3.52 dargestellt ist. Jede Assoziation kann mit einer Beschriftung versehen werden, die die Assoziation näher textuell beschreibt. Der schwarze Pfeil zeigt dabei die Leserichtung der Beschriftung an, also in diesem Fall „Punkt – kann Eckpunkt sein von – Dreieck“.

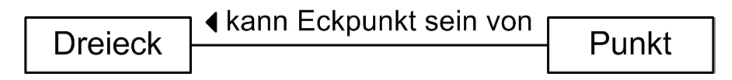


Abbildung 3.52: Erstes Klassendiagramm der Analyse

Ein Punkt kann Eckpunkt eines Dreiecks sein, muss es aber nicht. Ein Dreieck besteht aber stets aus genau drei Eckpunkten. Diese Abhängigkeiten werden als Multiplizitäten bezeichnet, die im Laufe der objektorientierten Analyse immer mehr herausgearbeitet werden. Abbildung 3.52 enthielt noch unspezifizierte Multiplizitäten. Die Möglichkeit, Diagramme mehr oder weniger detailliert anzugeben, ist typisch für die UML-Spezifikation und macht diese Sprache für eine iterative Vorgehensweise tauglich, bei der die zu erstellende Software in mehreren Stufen verfeinert wird.

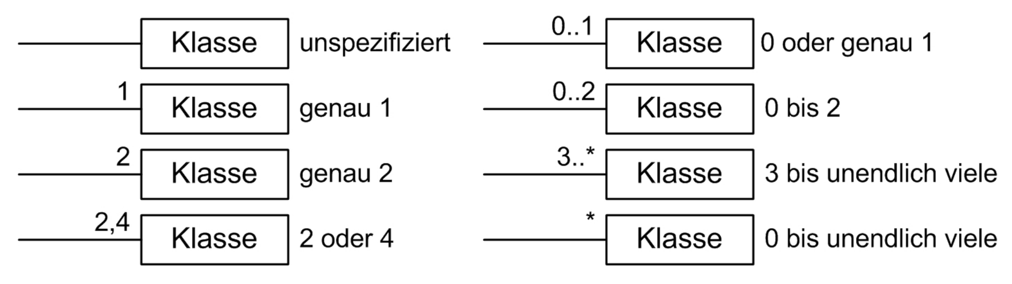


Abbildung 3.53: Angabe von Multiplizitäten

Wenn man die Analyse des Dreieck-Punkte-Beispiels weiter fortführt, so fällt die Tatsache auf, dass ein Dreieck aus Punkten besteht. Ein Punkt kann aber auch allein existieren oder gleichzeitig zu mehreren Dreiecken gehören. Dabei handelt es sich folglich um eine Aggregation. Diese Beziehung wird in der UML durch eine nichtausgefüllte Raute dar-

gestellt. Zusätzlich können noch die x- und y-Koordinaten als Eigenschaften der Punkte angegeben werden.

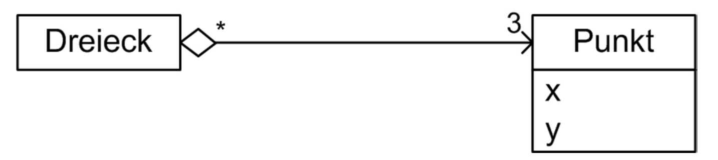


Abbildung 3.54: Fertiges Klassendiagramm der Analyse für Punkte und Dreiecke

Eine Komposition unterscheidet sich in der UML nur durch die Tatsache, dass in diesem Fall eine ausgefüllte Raute verwendet wird. Außerdem sollte man keine Multiplizität an einer Komposition angeben, da diese stets 1 beträgt. Genau so ist ja die Komposition definiert.

Als Beispiel für eine Komposition kann man eine Datei sehen, die sich stets in einem Verzeichnis befinden muss. Eine Datei ist ohne eine Verzeichnisstruktur nicht existenzfähig. Wird eine Datei gerade über ein Netzwerk übertragen, so verliert sie für eine gewisse Zeit ihre Existenz als Datei. Man könnte sie dann als Datenstrom oder als Datenpakete bezeichnen. Ein Verzeichnis kann jedoch auch leer sein, also keine Dateien enthalten.

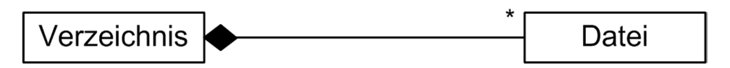


Abbildung 3.55: Beispiel einer Komposition

Ebenso werden fachliche Vererbungsbeziehungen in der objektorientierten Analyse festgehalten. Im folgenden Beispiel einer Universität wurden die Klassen *Angestellter*, *Student* und *Hilfskraft* mit einigen Eigenschaften und Methoden bereits ermittelt. Die Namen der Methoden werden separiert unter die Eigenschaften geschrieben. An dem Beispiel der Abbildung 3.56 fällt eine große Schnittmenge der Eigenschaften und der Methoden auf. Da eine doppelte Implementierung von identischem Quellcode zur besseren Wartbarkeit zu vermeiden ist, sollte in einem solchen Fall eine Vererbung genutzt werden.

Angestellter	Student	Hilfskraft
personalnr	matrikelnr	matrikelnr
name	name	name
anschrift	anschrift	anschrift
geburtsdatum	geburtsdatum	geburtsdatum
gehalt	immatrikulation	immatrikulation
bankverbindung		beschäftigungen
druckeAnschrift()	druckeAnschrift()	druckeAnschrift()
überweiseGehalt()	druckeAusweis()	druckeAusweis()
		druckeArbeitszeiten()

Abbildung 3.56: Klassen vor Einführung einer Vererbungshierarchie

Offensichtlich beinhalten alle Klassen die Eigenschaften *name*, *anschrift* und *geburtsdatum*, was auf eine Vererbungsstruktur deutet. Zusätzlich ist überall die Methode *druckeAnschrift()* vorhanden. Sowohl die Studenten, als auch die Hilfskräfte besitzen die Eigenschaften *matrikelnr*, *immatrikulation* und die Methode *druckeAusweis()*.

Der Unterschied zwischen diesen beiden Klassen besteht nur darin, dass eine Hilfskraft zusätzlich Beschäftigungen hat und eine Liste ihrer Arbeitszeiten drucken kann. Eine Hilfskraft ist also ein spezieller Student, sodass die Vererbung hier eindeutig festgehalten werden kann.

Eine Hilfskraft ist jedoch nicht fest angestellt. Ebenso ist ein Angestellter kein Student, da diese Klassen jeweils verschiedene Attribute besitzen, beispielsweise eine Personalnummer anstatt einer Matrikelnummer. Hier lässt sich also keine direkte Vererbung aufbauen. Die Lösung besteht darin, alle Gemeinsamkeiten der drei Klassen in einer abstrakten Oberklasse als Container für gemeinsame Eigenschaften und Methoden zusammenzufassen. So resultiert die Klassenhierarchie aus Abbildung 3.57. Abstrakte Klassennamen werden kursiv geschrieben oder um den Vermerk *{abstract}* ergänzt.

Wie es bei einer Vererbung typisch ist, werden alle Eigenschaften und Methoden der Oberklasse auf die Unterklasse mit vererbt. Wie Sie erkennen, ist keine mehrfache Deklaration mehr vorhanden. Den Vererbungspfeil können Sie mit der Phrase *ist ein* benennen. So ist ein Angestellter eine Person, ebenso wie ein Student. Eine Hilfskraft ist ein spezieller Student.

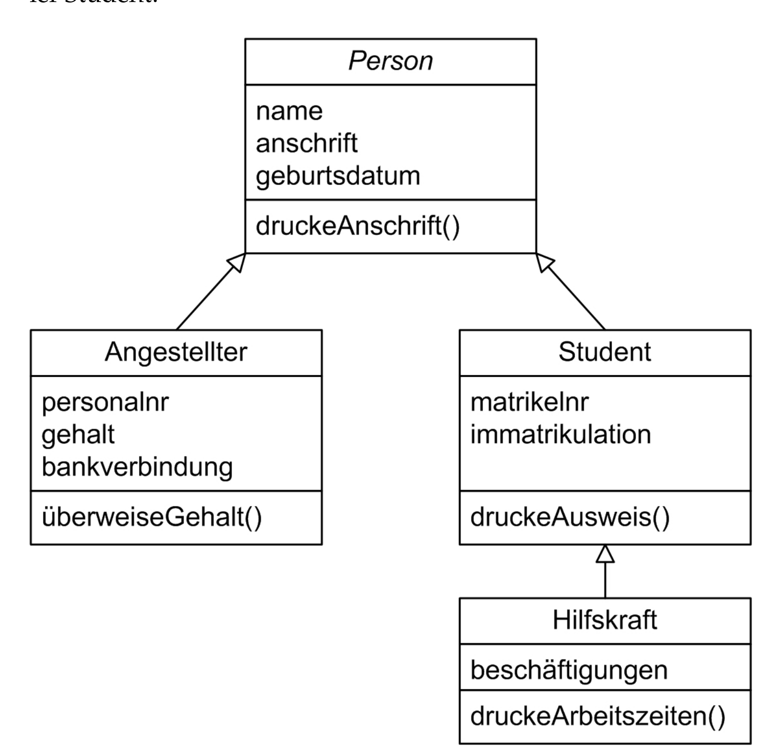


Abbildung 3.57: Resultierende Vererbungshierarchie

Ein sehr hilfreiches, jedoch zu selten eingesetztes Mittel der UML sind die Diskriminatoren, mit denen Sie spezifizieren, wie Sie eine Vererbung durchführen. Abbildung 3.58 zeigt, dass Sie Angestellte spezialisieren können nach Vollzeit- und Teilzeitkräften. Andererseits können Sie in Ihrer Anwendung auch eine Unterscheidung anhand der Tätigkeiten der Angestellten vornehmen. Beide Ideen sind sicherlich korrekt. Welche Art der Vererbung Sie letztlich wählen, hängt von der Art der Anwendung ab, die Sie erstellen wollen. Wenn Sie in Ihrer Anwendung eher Arbeitszeiten verwalten, so ist die erste Idee sinnvoller, bei einer Verwaltung von Berufsgruppen die zweite.

Es ist in der Analyse durchaus üblich, mehrere richtige Lösungen zu erstellen, die jedoch völlig verschiedene Ansätze verfolgen. Letztlich sollten Sie mit der Zeit erkennen, welcher Lösungsansatz der bessere ist.

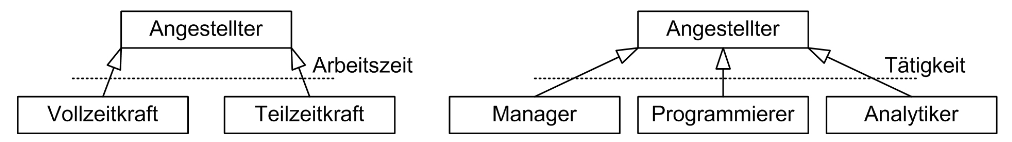


Abbildung 3.58: Vererbung mit Diskriminatoren

Wenn eine Assoziation zwischen zwei Klassen komplexer dargestellt werden muss als mit einer stichwortartigen textuellen Beschreibung, dann kann man dazu in der Analysephase eine eigene Klasse verwenden, die die Assoziation genauer beschreibt.

Nehmen wir als Beispiel an, Sie wollen die Beziehung zwischen einem Leser und einem Buch genauer beschreiben. Da es sich um eine Bibliothekssoftware handeln soll, sind Leser und Bücher über Ausleihen miteinander verbunden. Eine Ausleihe hat unter anderem ein Datum, zu dem der Leser das Buch ausgeliehen hat und die Möglichkeit, die Ausleihe ein- oder mehrfach zu verlängern. Wie die Assoziationsklasse letztlich umgesetzt wird, ist Aufgabe der Entwickler im Systemdesign.

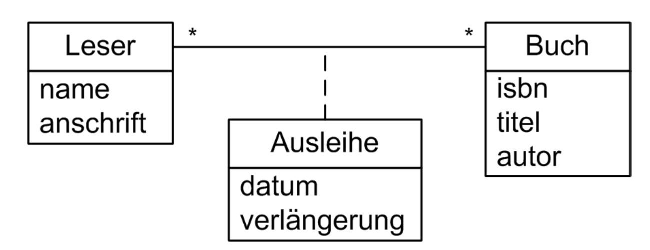


Abbildung 3.59: Beispiel einer Assoziationsklasse

Assoziationen sind auch zwischen mehr als zwei Klassen möglich. Diese Assoziationen werden als „n-äre Assoziationen“ bezeichnet. So wird die Beziehung zwischen einem Passagier, einem Flug und einem Sitzplatz über die Reservierung hergestellt. Das Erkennen solcher Zusammenhänge bildet den Kern der objektorientierten Analyse.

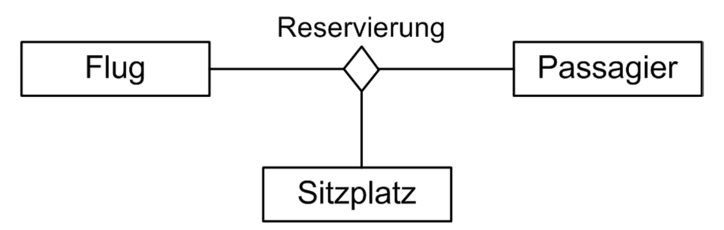


Abbildung 3.60: Beispiel einer n-ären Assoziation

Beispiel

Wie sind ein Kaufinteressent, ein Artikel und ein Verkäufer in der Modellierung miteinander verbunden? Die Antwort lautet: Über ein Verkaufsgespräch, das im Nachhinein protokolliert wird.

Eine weitere Besonderheit der objektorientierten Analyse sind reflexive Assoziationen, bei denen Objekte einer Klasse andere Objekte derselben Klasse kennen können. Dabei sind oft verschiedene Rollen von Bedeutung, deren Bezeichnung an die Assoziation geschrieben werden können.

Im Beispiel in Abbildung 3.61 sind sowohl die Chefs als auch deren Mitarbeiter Angestellte eines Unternehmens. Da Angestellte nur einmalig mit ihrer Personalnummer im System registriert sein sollen und ein Angestellter gleichzeitig Chef und Mitarbeiter sein kann, macht eine Aufspaltung der Klasse in die Klassen *Chef* und *Mitarbeiter* keinen Sinn.

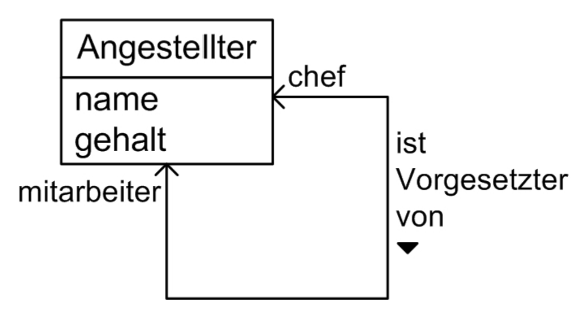


Abbildung 3.61: Beispiel einer reflexiven Assoziation

Wie würden Sie eine solche Beziehung realisieren? Im Quellcode bedeutet dies, dass die Klasse *Angestellter* zwei Datenfelder als Eigenschaften verwaltet. In dem ersten Feld *$istChefVon* werden die Referenzen auf andere Angestellte gespeichert, von denen dieser Angestellte der Vorgesetzte ist. Das zweite Feld *$istMitarbeiterVon* beinhaltet Referenzen auf Angestellte, denen dieser Angestellte weisungsbefugt ist. Generell kann eine reflexive Assoziation stets mit zwei Datenfeldern realisiert werden.

Profitipp

Versuchen Sie nicht, zwingend n-äre Assoziationen oder reflexive Assoziationen bei einer objektorientierten Analyse zu finden. Nur wenn diese besonderen Fälle wirklich von Bedeutung sind, sollten Sie sie auch verwenden. Ansonsten laufen Sie Gefahr, Ihre Problemstellung auf die UML zurechtzubiegen, anstatt die UML-Notation auf Ihr Problem anzuwenden.

Als abschließendes Beispiel der Analysephase wird nochmals das Beispiel der Seminarverwaltung aufgegriffen. Abbildung 3.62 zeigt zunächst ein Objektdiagramm. Das Verwaltungssystem verwaltet Seminare mit deren Terminen. Die Abbildung zeigt zwei konkrete Termine, wobei es bei einem Termin bereits zwei Anmeldungen von Kunden gibt. Das Anmeldungsobjekt verbindet also die Kunden mit einem Seminartermin. Zusätzlich werden jedem Termin ein Raum und ein Dozent zugeordnet.

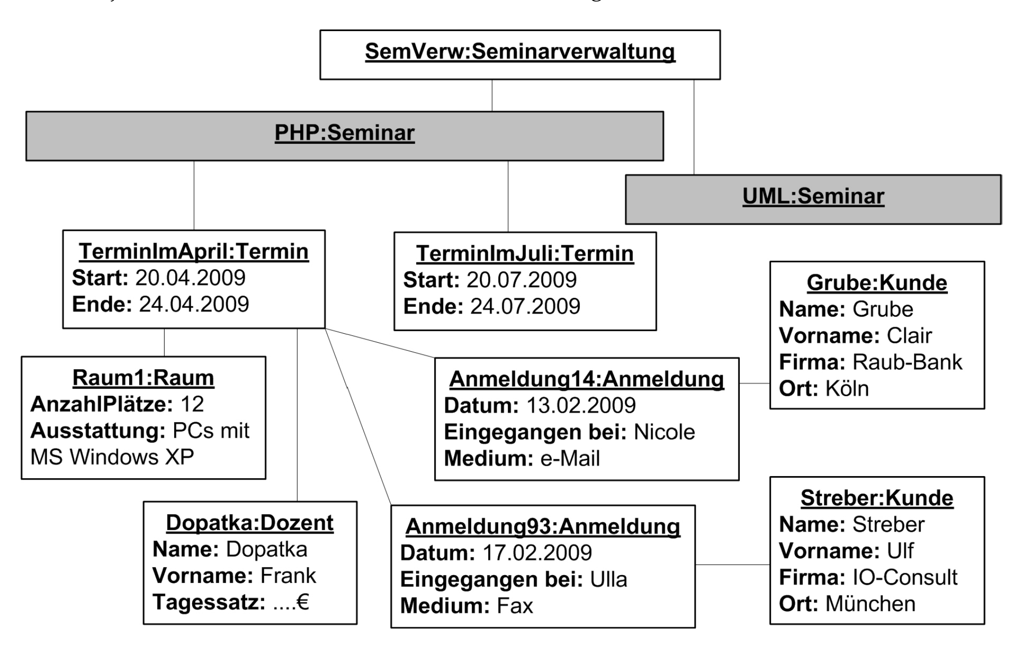


Abbildung 3.62: Objektdiagramm der Seminarverwaltung

Im nächsten Schritt wird nun aus diesem Objektdiagramm das Klassendiagramm der Analysephase abgeleitet. Die Namen der Klassen kann man bereits aus dem Objektdiagramm entnehmen. Sie lauten

- Seminarverwaltung
- Seminar
- Termin
- Raum

- Dozent
- Anmeldung
- Kunde

Die erste Version des Diagramms ist in Abbildung 3.63 dargestellt und benennt erst einmal die Beziehungen zwischen den Klassen textuell. Es kann in weiteren Schritten um Multiplizitäten, Aggregationen bzw. Kompositionen sowie um die Navigierbarkeiten erweitert werden.

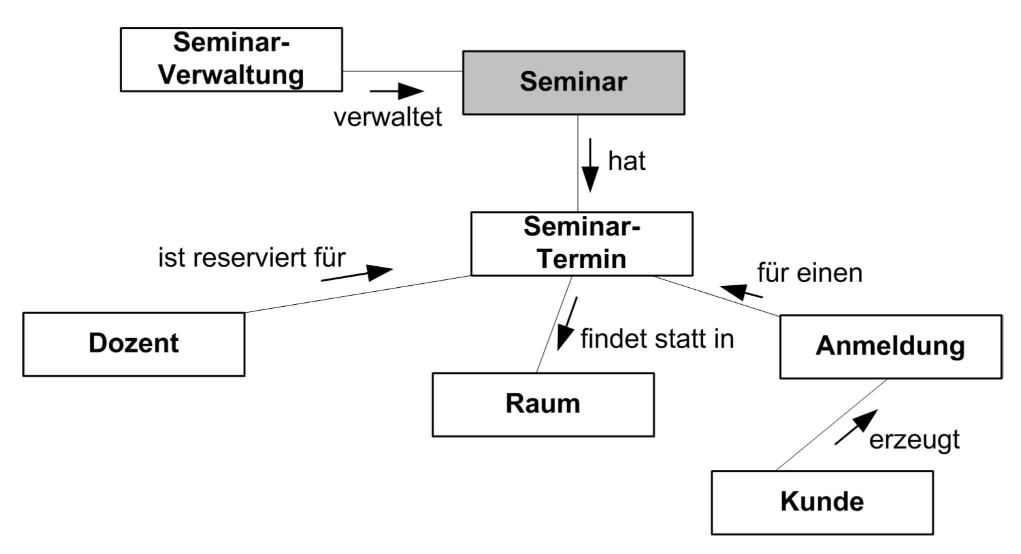


Abbildung 3.63: Erstes Klassendiagramm der Seminarverwaltung (Analysephase)

Klassen im objektorientierten Design

In der Designform beinhalten Klassendiagramme alle notwendigen Methoden und Eigenschaften. Es existieren bereits Modellierungswerkzeuge, die aus solchen Diagrammen Coderümpfe für verschiedene objektorientierte Programmiersprachen wie Java, C# und auch PHP generieren. Diese müssen dann vom Entwickler nur noch mit Funktionalität gefüllt werden.

Während sich die objektorientierte Analyse auf den Geschäftsprozess konzentriert und ihn in einem fachlichen Modell abbildet, fokussiert sich das objektorientierte Design auf das so genannte technische Modell. Dies ist die weitere Abstraktion des fachlichen Modells auf die Fähigkeiten einer objektorientierten Programmiersprache. Die hier erstellten Diagramme (zumeist Klassen-, Zustands- und Sequenzdiagramme) dienen den Entwicklern als direkte Vorlage für die objektorientierte Programmierung. Hier kann man erkennen, dass der Aufwand der Implementierung im Vergleich zur Analyse und Design geringer geworden ist, wie es im RUP-Modell beschrieben wurde. Dies gilt ebenso für die Verwendung agiler Methoden, die den Kunden durch das iterativ-inkrementelle Vorgehen stärker in den Entwicklungsprozess einbeziehen.

Diese Funktionalität wird in Sequenz- und Zustandsdiagrammen beschrieben, die die Interaktion von Objekten untereinander beschreiben. Diese Diagramme werden wiederum von den Anwendungsfall- und Aktivitätsdiagrammen abgeleitet.

Zunächst einmal muss in einem vollständigen Klassendiagramm angegeben werden, wie jede Eigenschaft und jede Methode außerhalb der Klasse von anderen Objekten gesehen wird. Es wurde bereits gesagt, dass Eigenschaften standardmäßig *private* sein sollen, um die Datenkapselung der Objektorientierung zu gewährleisten. Wenn Eigenschaften auch direkt aus Unterklassen angesprochen werden sollen, können diese als *protected* deklariert werden. Methoden sind Dienste der Klasse, die normalerweise von anderen Klassen verwendet werden sollen. Daher sind Methoden im Normalfall *public* einzustufen. Nur wenn es sich um Hilfsmethoden handelt, die innerhalb der Klasse verwendet werden, sollten diese *private* oder *protected* deklariert werden. Für diese Sichtbarkeiten besitzt die UML eine eigene Schreibweise.

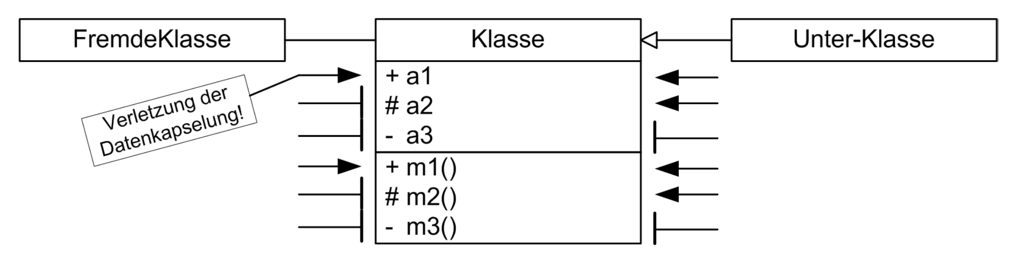


Abbildung 3.64: Sichtbarkeiten public, protected und private

Bei den Attributen sollten Sie in der Designphase zusätzlich die zu verwendenden Datentypen und falls nötig den Startwert angeben, der bei der Erzeugung eines neuen Objekts verwendet werden soll. Bedenken Sie, dass sich jedes Objekt zu jedem Zeitpunkt in einem gültigen Zustand befinden muss. So ist ein Bruch, der die Eigenschaften *zähler* und *nenner* hat, bei *nenner=0* ungültig und darf nach den Regeln der Mathematik nicht als Bruch bezeichnet werden. Achten Sie daher stets auf gültige Initialisierungswerte bereits in der UML-Darstellung!

Auch bei den Methoden sollten Sie Datentypen der Parameter benennen, die als Eingaben der Methode notwendig sind. Zusätzlich sollten Sie, falls vorhanden, den Datentyp des Rückgabewerts der Methode angeben. Dies ist auch dann sinnvoll, wenn Sie eine untypisierte Sprache wie PHP verwenden, bei der Sie den Variablen keine Datentypen zuweisen müssen.

In Abbildung 3.65 sehen Sie die UML-Darstellung der Klasse *Bruch* in Designform. Die Eigenschaften *zähler* und *nenner* sind nicht öffentlich zugänglich und werden auch beim Default-Konstruktor so initialisiert, dass ein gültiger Bruch entsteht. Die dritte Eigenschaft *anzahl* bezieht sich nicht wie die beiden anderen Eigenschaften auf jeden Bruch, sondern gehört zur Klasse *Bruch* selbst. Dies sieht man daran, dass diese Eigenschaft unterstrichen ist.

Zusätzlich zum Default-Konstruktor existiert ein zweiter Konstruktor, bei dem Sie den Zähler und Nenner als Ganzzahlen übergeben. Wie bei Konstruktoren üblich, existiert kein Rückgabewert.

Bruch
- zähler: int = 0 - nenner: int = 1 - anzahl: int = 0
+ Bruch(int,int) + add(int) + add(double) + add(Bruch) + isNull(): boolean + getAnzahl(): int

Abbildung 3.65: Eine Klasse in Designform

Als Dienste dieser Klasse werden 5 Methoden skizziert, die öffentlich verwendet werden können. Man kann zu einem Bruch eine Ganzzahl, eine Fließkommazahl – 0.34 entspricht 34/100 – und einen anderen Bruch addieren. Es gibt eine Methode, die prüft, ob der Bruch gerade den Wert 0 hat, also ob der Zähler=0 ist. Diese Methode gibt einen Wahrheitswert zurück. Die letzte Methode *getAnzahl()* gehört ebenso zur Klasse selbst und gibt den aktuellen Zählerstand der erzeugten Brüche, der in *anzahl* festgehalten wird, zurück. Der Zählerstand wird bei jedem Aufruf des Default-Konstruktors inkrementiert und bei jedem Destruktoraufruf dekrementiert.

Im nächsten Beispiel existiert, wie bereits bei der Person in der Analysephase, eine abstrakte Klasse, von der man keine Objekte anlegen kann. Diese Klasse *Tier* besitzt eine abstrakte Methode *gibLaut()*. Wenn man von einer Unterklasse von *Tier* Objekte anlegen will, muss man diese abstrakte Methode mit Quellcode füllen. Die Unterklassen *Katze* und *Hund* tun dies. Zusätzlich definieren sie noch die öffentlichen Methoden *miauen()* bzw. *bellen()*. Die Methode *gibLaut()* beinhaltet dann intern nur den Aufruf von *miauen()* bzw. *bellen()*. So kann ein Datenfeld von Tieren erstellt werden, in das man sowohl Katzen als auch Hunde ablegen kann. Auf jedem Tier in dem Feld kann man dann *gibLaut()* aufrufen und man hört entweder ein Miauen oder ein Bellen (Abb. 3.18). Dies entspricht der Anwendung der Polymorphie.

Jedem Hund kann man zusätzlich einen Stock geben. Der Hund kennt dann seinen Stock. Der Stock selbst ist dumm und hat weder eigene Eigenschaften noch Methoden. Ein Stock kann auch ohne Hund existieren, wenn er im Wald herumliegt. Außerdem existiert auch ein Hund, ohne einen Stock zu besitzen. Wie Sie sehen, lässt sich mit der UML die gesamte reale Welt abbilden.

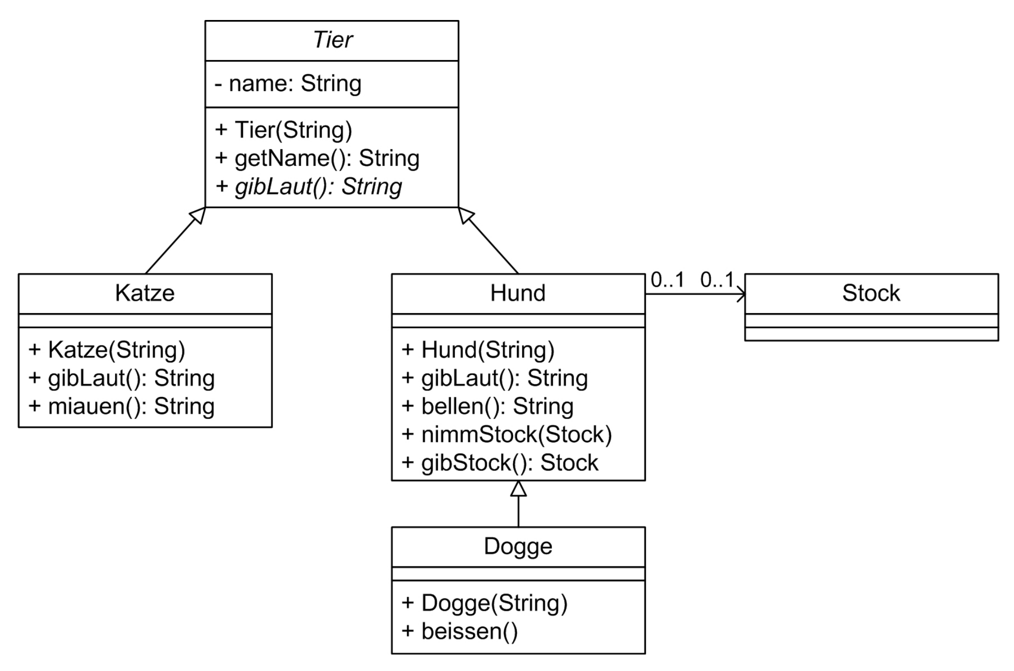


Abbildung 3.66: Klassengeflecht in Designform mit Vererbungshierarchie

Da ein Hund auch ein Tier ist, kann er folgende Methoden ausführen:

- *Hund(String)* als Default-Konstruktor, der wiederum *Tier(String)* aufruft, um den Namen des Tieres zu speichern
- *getName()* aus der Klasse *Tier*
- *gibLaut()* aus der Klasse *Hund*
- *bellen()* aus der Klasse *Hund*
- *nimmStock(Stock)* aus der Klasse *Hund*, wobei der Hund ein Stockobjekt entgegennimmt
- *gibStock()* aus der Klasse *Hund*, bei der der Hund sein Stockobjekt wieder abgibt

Eine Dogge ist ein spezieller Hund, der auch beißen kann. Da eine Dogge auch ein Hund ist, kann sie auch einen Stock kennen und alle Methoden eines Hundes ausführen. Eine Katze ist zwar auch ein Tier wie ein Hund, kann aber keinen Stock kennen.

Abschließend wird noch vorgestellt, wie man das Konzept der Interfaces in UML-Klassendiagrammen darstellen kann. Da ein Interface ausschließlich Funktionalität beschreibt, beinhaltet es lediglich Methodendefinitionen, die jedoch noch nicht implementiert sind. Diese Methodendefinitionen können mit der abstrakten Methode *gibLaut()* der Tierklasse verglichen werden. Im Beispiel der Abbildung 3.67 wird ein allgemeines Datenzugriffsinterface beschrieben, das zunächst unabhängig von einer Datenbank ist.

Die Klasse *DZ_MySQL* implementiert nun dieses Interface. Eine Klasse kann auch mehrere Interfaces implementieren. Dadurch wird die Klasse gezwungen, alle im Interface definierten Methoden zu implementieren. Die Klasse bietet also einen Datenzugriff auf eine MySQL-Datenbank an. Genauso können Sie einen Zugriff für eine Oracle-Datenbank oder auf ein Dateisystem implementieren. Wenn jemand einen solchen Zugriff verwenden will, kann er das Interface verwenden und damit sicher sein, dass genau diese Methoden bereit stehen.

Bei großen Projekten ist es üblich, dass die Interfaces in einer frühen Projektphase definiert und dann vom Projektmanagement verwaltet werden. Auf diese Weise kann ein Entwicklertam den Datenzugriff implementieren, während ein anderes Team bereits mit einer Dummy-Klasse auf Basis des Interfaces arbeitet und die Fachlogik darauf aufbauen. Außerdem kann die Fachlogik Referenzen auf dieses Interface besitzen und so die Implementierungen leicht auswechseln. Dadurch ist die Datenbank leicht austauschbar und ein weiterer Schritt in eine komponentenbasierte Anwendung erfolgt.

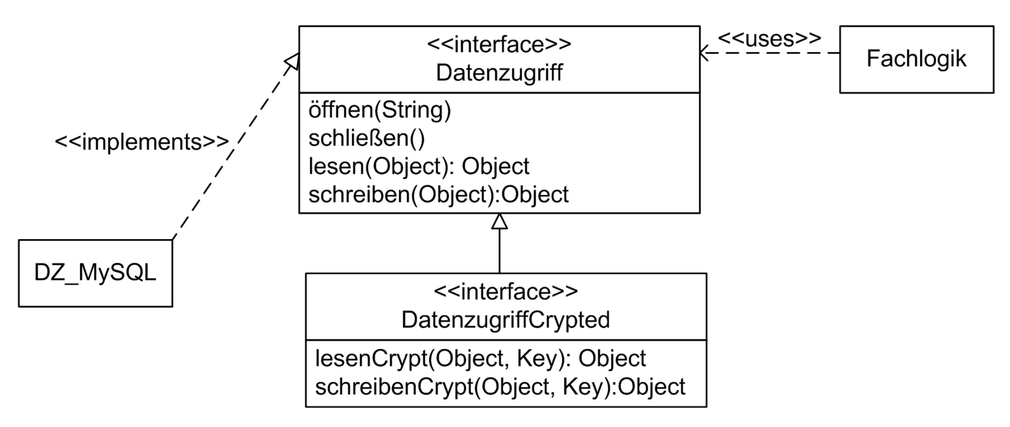


Abbildung 3.67: Klassen und Interfaces

Profitipp

Bei kleinen Projekten kann die Verwendung von zu vielen Interfaces zu einem „Over-Design“ führen und den Quellcode unnötig vergrößern.

Bestehende, bereits in Projekten verwendete Interfaces sollten nicht verändert werden. Stattdessen besteht die Möglichkeit, auch Interfaces voneinander vererben zu lassen. So erweitert das Interface *DatenzugriffCrypted* das Interface *Datenzugriff* um die Option zum Verschlüsselten Lesen und Schreiben, wobei jeweils Schlüsselobjekte übergeben werden. Wenn eine Klasse das neue Interface implementieren will, müssen alle 6 Methoden implementiert werden. Für Kunden, die weniger Wert auf eine verschlüsselte Übertragung legen und für Altkunden steht das alte Interface *Datenzugriff* mit seinen Implementierungen weiterhin unverändert zur Verfügung.

Hinweis

Eine gute Übung besteht darin, aus dem Objekt- und Klassendiagramm der Analysephase aus der Seminarverwaltung ein Klassendiagramm der Designphase zu erstellen.

Zustände in einem Objekt: Zustandsdiagramme

Zustandsdiagramme stellen endliche Automaten in einer UML-Sonderform grafisch dar. Ein Zustandsdiagramm zeigt eine Folge von Zuständen, die ein Objekt im Laufe seines Lebens einnehmen kann. Außerdem gibt es an, aufgrund welcher Stimuli Zustandsänderungen stattfinden. Stimuli sind Ereignisse von außen, die das Objekt zu einer Aktion anregen. Da ein Objekt ausschließlich über Methodenaufrufe angesprochen wird, sind diese Ereignisse Methodenaufrufe von anderen Objekten oder von einem Anwender, der beispielsweise eine Schaltfläche betätigt.

Damit beschreibt ein Zustandsdiagramm eine hypothetische Maschine (endlicher Automat), die sich zu jedem Zeitpunkt in einer Menge endlicher Zustände befindet. Die Automatentheorie ist eine eigene Mathematik. Diagramme zur Darstellung von so genannten Moore- und Mealy-Automaten existieren bereits seit 1950. Die UML hat auch diese Notationen lediglich vereinheitlicht und in ihre eigene Sprache integriert.

Ein Klassendiagramm im Design stellt alle Eigenschaften, Methodenaufrufe und Bekanntschaften von Klassen untereinander dar. Zu Beginn dieses Kapitels wurde bereits gezeigt, dass die Modellierung der Klassendiagramme mit der Entity-Relationship-Modellierung aus der Datenbankentwicklung verglichen werden kann. Klassendiagramme beschreiben also in erster Linie statische Datenstrukturen und Komponenten der zur erstellenden Anwendung.

Jede Methode einer Klasse (außer die Konstruktoren) kann prinzipiell zu jedem Zeitpunkt nach der Erstellung eines Objekts aufgerufen werden. Dies ist jedoch nicht immer sinnvoll. Wenn Sie beispielsweise einen Warenkorb modellieren, können Sie nur dann zur Kasse gehen, wenn auch Artikel im Korb enthalten sind. Außerdem können Sie nur dann Artikel entfernen, wenn sich etwas im Korb befindet. Ein Zustandsdiagramm bezieht sich also immer nur auf genau eine Klasse.

Profitipp

Für ein gutes objektorientiertes Design ist es sinnvoll, für größere Klassen, deren Objekte in ihrer Lebenszeit mehrere Zustände durchlaufen können, Zustandsdiagramme anzufertigen. Dies gilt auch dann, wenn nicht in jedem Zustand jede Methode einer Klasse aufgerufen werden darf.

Die Zustände in einem Zustandsdiagramm werden durch Rechtecke mit abgerundeten Ecken – in anderen Diagrammformen außerhalb von UML häufig auch Kreise, Ellipsen oder einfache Rechtecke – dargestellt. Die möglichen Zustandsübergänge werden durch Pfeile zwischen den Zuständen symbolisiert. Die Pfeile sind mit den Ereignissen beschriftet, die zu dem jeweiligen Zustandsübergang führen können.

In der zweiten Version der UML wurden die Zustandsdiagramme in ihrer Notation erweitert. So kann bei jedem Zustandswechsel durch einen Methodenaufruf eine Bedingung angegeben werden, die für diesen Aufruf erfüllt sein muss. So kann man die Bedingung später im Quellcode der Methode direkt prüfen, bevor mit dem Zustandswechsel fortgefahren wird. Hier erkennen Sie bereits, dass sich detaillierte Zustandsdiagramme bereits dem zu erstellenden Quellcode nähern. Zusätzlich kann bei Bedarf eine Aktion definiert werden, die während des Zustandswechsels ausgeführt werden soll.

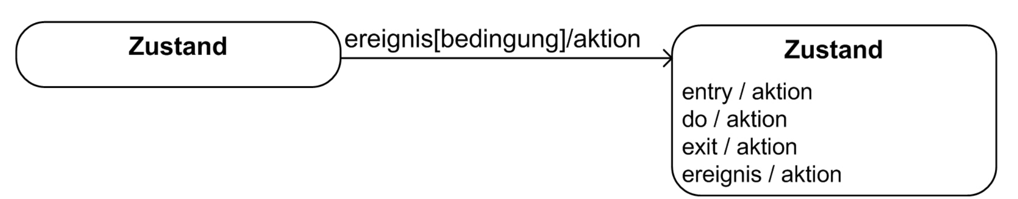


Abbildung 3.68: Zustände und Zustandswechsel

Außerdem können Sie innerhalb eines Zustands definieren, welche Aktion beim Eintritt (*on entry*) und beim Verlassen (*on exit*) dieses Zustands ausgeführt werden soll. Des Weiteren können Sie eine oder mehrere Aktionen definieren, die ausgeführt werden, während Sie sich in diesem Zustand befinden (*do*).

Um Schlingen im Zustandsdiagramm zu vermeiden, können Sie innerhalb des Zustands auch eine Liste von Ereignissen definieren, die Aktionen auslösen, während man sich in diesem Zustand befindet.

Wenn Sie Zustandsdiagramme erstellen, sollten Sie einmalig zu Beginn textuell definieren, was geschehen soll, wenn ein Methodenaufruf – also ein Stimulus – eintritt, der im aktuellen Zustand nicht definiert ist. In der Regel definieren Sie für diesen Fall eine Fehlerausgabe und verbleiben in diesem Zustand.

Die Notation des Starts sowie des regulären und irregulären Endes wurde von den Aktivitätsdiagrammen übernommen. Es ist üblich, bei Zustandsdiagrammen den ersten Zustand *initialisiert* oder *idle* zu erzeugen, der direkt nach der Erzeugung des Objekts eintritt und das sinnvoll initialisierte Objekt beschreibt.

Start
reguläres Ende
irreguläres Ende

Abbildung 3.69: Start, reguläres und irreguläres Ende des Zustandsautomaten

Im ersten Beispiel wird das Zustandsdiagramm eines Interfaces beschrieben, das das Protokoll einer sinnvollen Realisierung dieses Interfaces angibt. Das Interface definiert die Methoden einer Flugreservierung. Jemand, der dieses Interface implementiert, muss die folgenden Methoden ausprogrammieren:

- reservieren
- stornieren
- buchen

Das Zustandsdiagramm beschreibt nun, in welcher Reihenfolge diese Methoden aufgerufen werden dürfen und in welchem internen Zustand sich die Flugreservierung dann jeweils befindet. Diese Dynamik kann in einem Klassendiagramm nicht dargestellt werden.

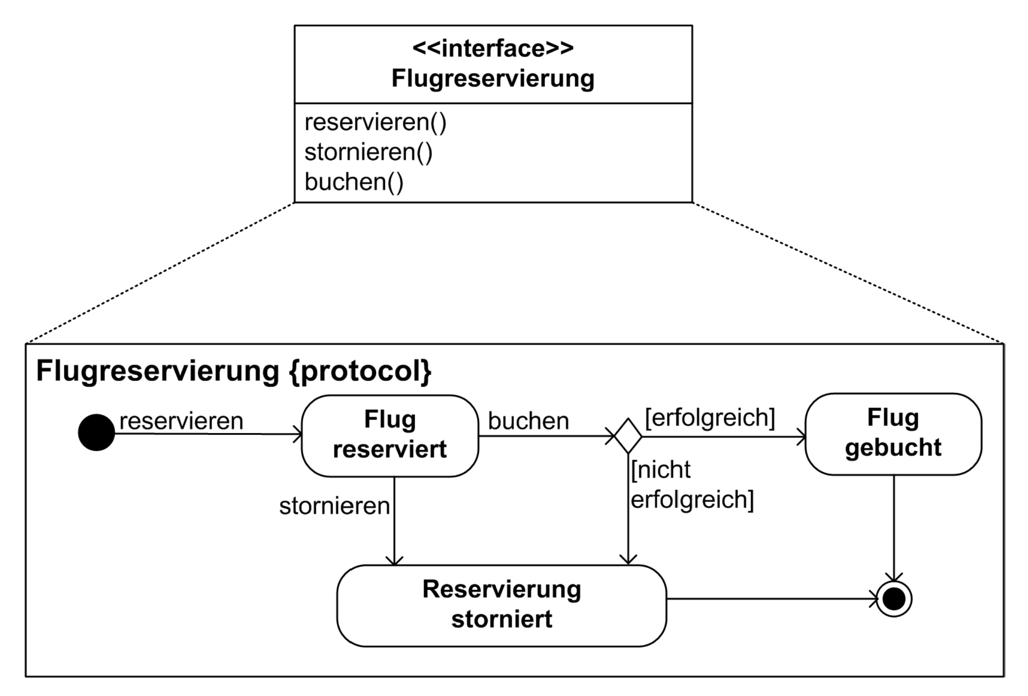


Abbildung 3.70: Protokoll einer Flugreservierung

Nach der Erzeugung eines Reservierungsobjekts, das diesem Interface gehorcht, können Sie nur die Methode *reservieren* aufrufen. Im Anschluss daran ist der Flug reserviert. Eine Reservierung kann entweder verbindlich gebucht oder storniert werden. Es kann sein, dass die Buchung nicht erfolgreich ist, weil der Flug ggf. bereits schon ausgebucht ist und eine Reservierung nicht den Anspruch auf eine Buchung erfüllt.

Auffallend ist auch bei diesem Reservierungsprotokoll, dass ein einmal gebuchter Flug nicht mehr storniert werden kann. Bei diesem Anbieter sind Sie also an eine einmal getätigte Buchung gebunden.

Profitipp

Der große Vorteil eines Zustands-Diagramms besteht darin, dass Sie eine Klasse gegen diesen Automaten leicht testen können, indem Sie alle möglichen Zustandsübergänge automatisiert durchtesten und mit einer zusätzlichen Methode *getZustand()* den aktuellen Ist-Zustand mit dem Soll-Zustand aus dem Diagramm vergleichen. Im nächsten Schritt können Sie bewusst zufallsbasiert irreguläre Methodenaufrufe in jedem Zustand auslösen, um die Reaktion des Objekts zu testen.

Das Beispiel der Abbildung 3.71 zeigt das Zustandsdiagramm eines Seminartermins auf Meeresspiegelebene. Nachdem der Termin angelegt wurde und sich noch niemand zu diesem Termin angemeldet hat, kann man die Seminaragenda und weitere Daten zu diesem Termin noch beliebig ändern.

Es kommt vor, dass sich niemand zu diesem Termin anmeldet. In diesem Fall entfällt der Termin und das Terminobjekt wird gelöscht.

Sobald sich der erste Kunde anmeldet, wird das Seminar zu diesem Termin in den Zustand *buchend* überführt. In diesem Status kann das Seminar zwar noch ausfallen, weil beispielsweise der Dozent erkrankt ist. In diesem Fall geht der Termin in den Zustand *storniert* über. Sobald dieser Zustand eintritt, werden alle bereits angemeldeten Teilnehmer benachrichtigt und über alternative Termine informiert.

In einem buchenden Seminar können sich weitere Teilnehmer an- bzw. abmelden. Wenn die Obergrenze der möglichen Teilnehmer erreicht ist, wechselt der Termin in den Zustand *ausgebucht*, in dem keine neuen Anmeldungen zugelassen werden. Auch ein ausgebuchter Termin kann aufgrund einer Krankheit des Dozenten noch ausfallen. Wenn sich bei einem ausgebuchten Seminartermin jemand wieder abmeldet, geht der Termin wieder in den Zustand *buchend* über, in dem neue Anmeldungen möglich sind.

Abbildung 3.71 zeigt auch die Möglichkeiten eines Zustandsdiagramms, Zeitbedingungen einzufügen. Über den Ausdruck *after(Seminarende)* wechselt ein Seminartermin vom Zustand *buchend* selbstständig in den Zustand *durchgeführt*, wenn es nicht ausgefallen ist. Sowohl durchgeführte als auch stornierte Seminartermine werden zur Rechnungsstellung und für statistische Zwecke archiviert.

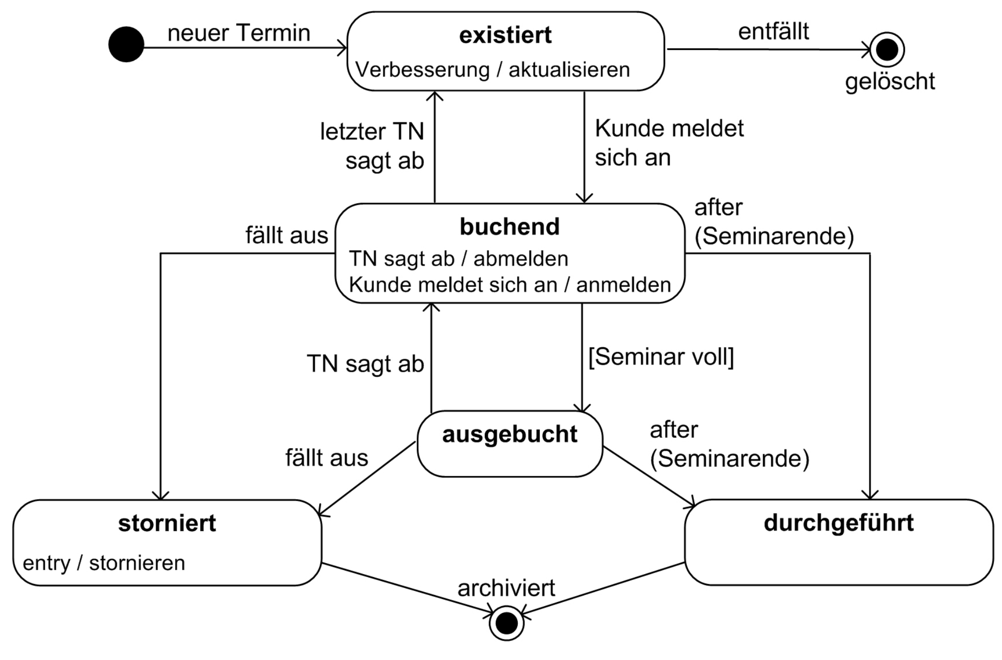


Abbildung 3.71: Zustandsdiagramm eines Seminartermins

Meinung

Ein präzise und vollständig ausformuliertes Zustandsdiagramm ist für den Entwickler eine große Hilfe, das erstellte technische Modell zu implementieren. Im Gegensatz zu Systemanalytikern, die mit dem Kunden aus dem Geschäftsprozess weichere Formulierungen zur Beschreibung der zu erstellenden Anwendung ausarbeiten, können Zustandsdiagramme mit klaren Formulierungen und Übergängen versehen werden, die sich leicht in einer Programmiersprache umsetzen lassen.

Das nächste Beispiel eines Zustandsdiagramms zeigt das Modell eines Getränkeautomaten auf Meeresspiegelebene, wie ein Anwender den Automat typischerweise bedient.

Nach dem Einschalten initialisiert sich der Automat mit seiner Getränkekühlung, seinem Füllstand der einzelnen Getränkefächer, dem Münzautomat usw. Der Automat bleibt solange in diesem Zustand, bis ein Kunde die erste Münze einwirft. Daraufhin wechselt er in den Zustand *kassierend*. Dort können weiterhin beliebig viele Münzen eingeworfen werden.

Wenn der Kunde ein Getränk wählt, wechselt der Automat in einen temporären Zustand, in dem das entsprechende Getränkefach auf Inhalt und das Wechselgeld geprüft wird. Gleichzeitig zeigt der Automat den Preis für das gewählte Getränk digital an. Ist der Behälter leer, wird dies angezeigt und der Automat bleibt im Zustand *kassierend*. Der Kunde kann dann ein anderes Getränk wählen oder den Vorgang abbrechen, wobei er sein eingezahltes Geld zurück erhält. Der Automat bleibt auch in dem Zustand, wenn noch nicht genügend Geld eingeworfen wurde.

Ist das eingezahlte Geld ausreichend, so wechselt der Automat in den temporären Zustand *Wechselgeld ausgeben*, wenn der Kunde noch Geld zurückbekommt. Ansonsten wird direkt die Mechanik des Getränkeauswurfs aktiviert und der Automat wechselt wieder zurück in den *bereit*-Zustand.

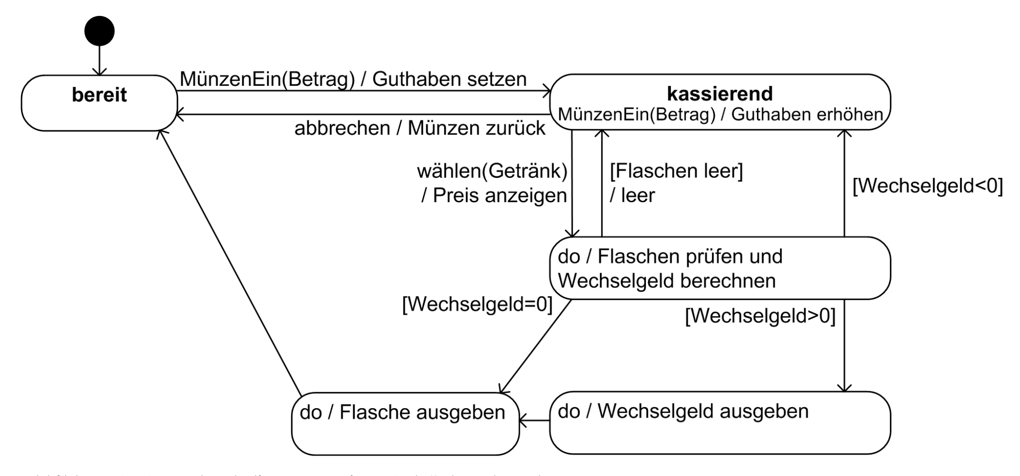


Abbildung 3.72: Zustandsdiagramm eines Getränkeautomaten

Meinung

Zustandsdiagramme können in einer kompakten Darstellung sehr viele Informationen enthalten und sehr aussagekräftig sein. Der Informationsgehalt von grafischen Anwendungsfalldiagrammen ist dagegen wesentlich geringer. Die Ursache liegt unter anderem darin, dass man sich erst langsam an die Anforderungen herantastet.

Das letzte Beispiel eines Zustandsdiagramms in Abbildung 3.73 modelliert einen Server des Onlinebankings auf Fischebene. Diese Darstellung geht also schon etwas tiefer ins Detail, als ein gewöhnlicher Nutzer dies wahrnimmt.

Der Server ist zunächst in einem betriebsbereiten Zustand, in dem er auf Verbindungen vom Client wartet.

Wenn sich ein Rechner mit dem Server verbindet, so ist dies nur möglich, wenn man sich auf ein verschlüsseltes SSL-Protokoll einigt. Danach ist der Client mit dem Bankserver verbunden.

Die Abbildung zeigt die Möglichkeit, innerhalb des globaleren Zustands *verbunden* Unter-Zustände zu definieren, die innerhalb dieses Zustands durchlaufen werden. Im nächsten Schritt sieht der Benutzer ein (HTML-)Formular, bei dem er seine Kontonummer und seine PIN eingeben muss. Die Daten werden dann vom (PHP-)Server geprüft.

Waren die Daten korrekt, gilt der Benutzer für den Server als authentifiziert und darf Banktransaktionen durchführen. Ansonsten wird die Verbindung abgebrochen und der Login-Vorgang aufgezeichnet. Auch nach einem Logout-Vorgang wird die Verbindung wieder unterbrochen.

Jede Transaktion wird von einem unabhängigen zweiten System parallel protokolliert. Wie bei Aktivitätsdiagrammen können auch bei Zustandsdiagrammen parallel ausgeführte Vorgänge modelliert werden. Die Durchführung von Banktransaktionen ist so komplex, dass es in einem separaten Zustandsdiagramm, das hier nicht dargestellt wird, festgehalten ist. Darin werden typische Anwendungsfälle ausgeführt, die ein Kunde wünscht, wie Kontostände einzusehen, Überweisungen zu tätigen, Daueraufträge einzurichten, Zugriff auf sein Aktiendepot zu nehmen oder sein Handy aufzuladen.

In sehr seltenen Fällen können Probleme auftreten, während ein Benutzer mit dem Server verbunden ist. In diesem Fall wird die Verbindung unterbrochen, protokolliert und an einen Administrator des Systems übergeben.

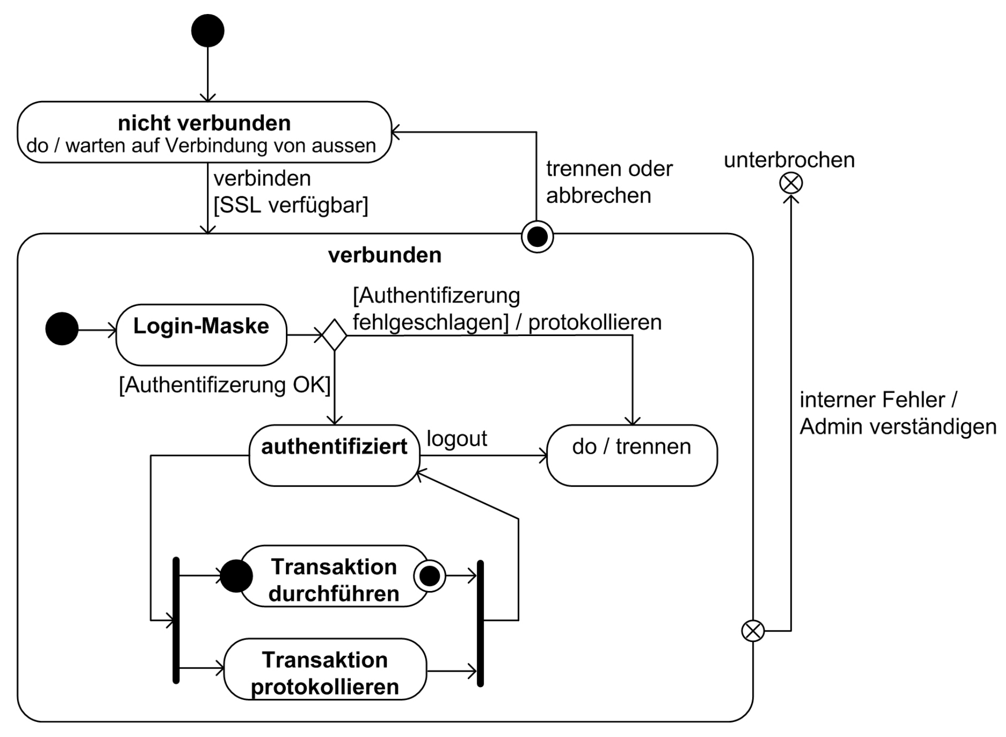


Abbildung 3.73: Zustandsdiagramm eines Bankservers

Abläufe im technischen Modell: Sequenzdiagramme

Bereits bei den Aktivitätsdiagrammen können fachliche und technische Abläufe von der Wolken- bis zur Muschelebene dargestellt werden, wobei Workflows in den Szenarien oder auch Quellcodeflüsse abgebildet werden können.

Objekte agieren durch Methodenaufrufe miteinander. In der objektorientierten Analyse, in der Aktivitätsdiagramme typischerweise zum Einsatz kommen, sind diese Objekte noch nicht bekannt. Die Analyse dient schließlich dazu, Objekte und Klassen zu ermitteln.

Sequenzdiagramme stellen eine zusätzliche Sichtweise dar, um den zeitlichen Verlauf von Interaktionen der Objekte festzuhalten. Zusätzlich kann dargestellt werden, wann Objekte erzeugt bzw. zerstört werden und wann sie aktiv sind, also Methoden abarbeiten und Methoden von anderen Objekten aufrufen. Der Auslöser eines Methodenaufrufs ist dabei ein Akteur, der in einem Anwendungsfalldiagramm identifiziert wurde. An dieser Stelle ist die Vernetzung und Überdeckung der einzelnen Diagrammarten gut zu erkennen.

Ein Sequenzdiagramm dient demnach zur Darstellung genau eines Szenarios, das in der Analyse spezifiziert wurde.

Hinweis

Es ist nochmals darauf hinzuweisen, dass die Phasen der fachlichen Analyse, des technischen Designs und der Implementierung nicht als einmalig aufeinanderfolgende Schritte, sondern als iterativ-inkrementeller Prozess zu sehen sind.

Da Klassen lediglich Baupläne für Objekte sind, können sie nicht direkt miteinander kommunizieren. In einem Sequenzdiagramm stehen Objekte im Vordergrund. Abbildung 3.74 zeigt die Erzeugung des Objekts *einX* der Klasse *X*, also den Aufruf des Konstruktors. Die Schreibweise wurde aus den Objektdiagrammen übernommen und ist somit innerhalb der UML konsistent. Auch anonyme Objekte können dargestellt werden.

Die gestrichelte Linie wird als Lebenslinie bezeichnet und der Balken als Zeitraum, in dem das Objekt aktiv Methoden bearbeitet. Dieser Balken wird „Aktivierungsbalken“ genannt. In dieser Zeit besitzt das Objekt den Kontrollfluss und ist aktiv an Interaktionen beteiligt. Dieser Zustand wird als „Focus of Control“ bezeichnet.

Das abschließende Kreuz stellt den Destruktoraufruf und damit die Zerstörung des Objekts im Speicher dar.

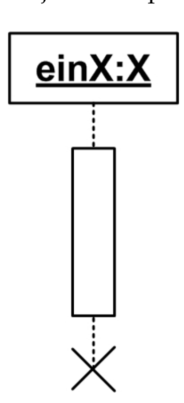


Abbildung 3.74: Lebenslinie eines Objekts im Aktivitätsdiagramm

Ein Sequenzdiagramm beschreibt das Verhalten eines Systems, indem es die zeitliche Ordnung von Methodenaufrufen spezifiziert. Diese Aufrufe werden auch als Botschaften von außen oder als Stimuli in Zustandsdiagrammen bezeichnet.

Nicht der präzise Zeitpunkt, wann solch ein Ereignis auftritt, ist ausschlaggebend, sondern welche Methodenaufrufe vor und welche nach einem bestimmten Ereignis auftreten müssen.

Die Zeitachse verläuft in einem Sequenzdiagramm von oben nach unten, sollte aber nicht als absolute Zeit verstanden werden. Es können lediglich kausale zeitliche Abhängigkeiten von Aufrufen festgehalten werden. So tritt E1 aus Abbildung 3.75 nach S1 auf, weil das Empfangs- immer nach dem Sendeereignis vorkommt. Analog tritt E2 nach S2 auf. S2 tritt nach S1 auf, weil S2 unter S1 gezeichnet ist

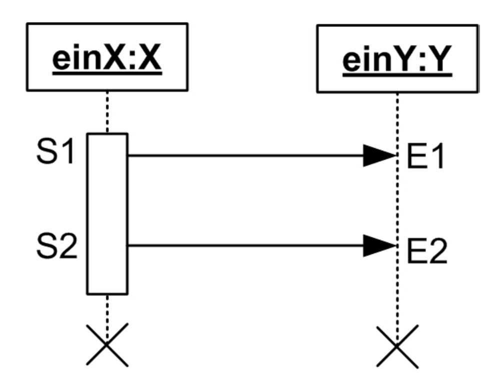


Abbildung 3.75: Kausale zeitliche Abhängigkeiten

Auch Verzweigungen, also alternative Abläufe, können in Sequenzdiagrammen dargestellt werden. Diese Darstellung kann jedoch leicht zu sehr unübersichtlichen großen Diagrammen führen und ist daher nur in begrenztem Maße anzuwenden. Generell sollte ein Sequenzdiagramm nur genau ein Szenario darstellen. Alternative Abläufe sind, wenn möglich, in getrennten Diagrammen festzuhalten. Abbildung 3.76 zeigt das Verhalten eines Objekts *einX* in Abhängigkeit einer Bedingung B. Ist B wahr, so wird die Methode *m()* auf dem Objekt *einY* angesprochen, ansonsten die Methode *n()* auf dem Objekt *einZ*. Diese Methoden werden dann entsprechend abgearbeitet und lösen unter Umständen weitere Interaktionen aus. Die UML-Syntax zur Beschreibung einer Bedingung ist konsistent zu den Aktivitäts- und Zustandsdiagrammen.

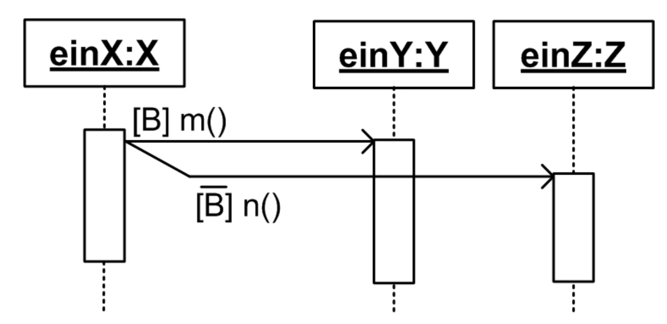


Abbildung 3.76: Verzweigung mit verschiedenen Objektzugriffen

Eine andere Art der Darstellung einer Verzweigung ergibt sich, wenn aufgrund der Bedingung eine andere Methode auf demselben Objekt aufgerufen wird. Da jeder Methodenaufruf Einfluss auf den Zustand eines Objekts haben kann, erhält das Objekt *einY* eine andere Historie in Abhängigkeit der Bedingung B. Dies wird durch eine verzweigte Lebenslinie dargestellt. Die alternativen Lebenslinien können wiederum alternative Methodenaufrufe auf anderen Objekten nach sich ziehen.

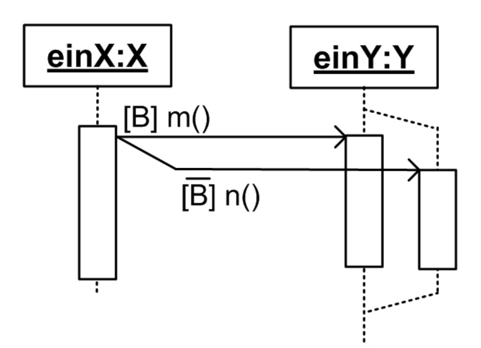


Abbildung 3.77: Verzweigung mit alternativen Lebenslinien

Eine weitere Besonderheit der Sequenzdiagramme besteht darin, synchrone von asynchronen Aufrufen zu unterscheiden, indem diese beiden Arten von Aufrufen mit unterschiedlichen Pfeilen dargestellt werden.

Abbildung 3.78 zeigt drei Methodenaufrufe. Der erste Aufruf ist synchron, benötigt jedoch kaum Rechenzeit. Es existiert kein Rückgabewert, der für das auslösende Objekt *einX* von Bedeutung ist.

Der zweite Aufruf ist asynchron, was man durch die anders geformte Pfeilspitze erkennen kann. In den Versionen 1.x der UML wurde der Aufruf durch eine offene halbe Pfeilspitze nach oben dargestellt, seit der Version 2 durch eine vollständige offene Pfeilspitze.

Dieser asynchrone Aufruf besitzt ebenfalls keinen Rückgabewert und ist daher einfach zu implementieren. Was jedoch macht einen asynchronen Aufruf aus? Bei einem asynchronen Methodenaufruf blockiert das aufrufende Objekt nicht, während die aufgerufene Methode längere Zeit abgearbeitet wird. Die aufrufende Methode wird also (quasi-)parallel zu der aufrufenden Methode abgearbeitet. Dies lässt sich mit Multi-Threading oder Multi-Processing realisieren, ist also technisch aufwändig.

Ein Standardbeispiel für einen asynchronen Vorgang ist die Arbeit in einem Textverarbeitungsprogramm, wenn Sie sich für das Drucken Ihres großen Dokuments entscheiden. Der Druckvorgang wird gestartet und gleichzeitig können Sie an Ihrem Text weiterarbeiten. Bei einem Dokument von beispielsweise 100 Seiten ist es heutzutage nicht akzeptabel, wenn Sie Ihre Anwendung 2 Minuten lang nicht bedienen können.

Der dritte Aufruf aus Abbildung 3.78 zeigt einen blockierenden synchronen Aufruf, der auf einen Rückgabewert wartet. Das Objekt *einX* hat zu diesem Zeitpunkt zwar den „Focus of Control", wartet aber auf die Rückgabe von *einY*. Diese inaktive Wartezeit wird durch einen gestrichelten Balken dargestellt. Beachten Sie auch die Darstellung des Pfeils, der den Rückgabewert symbolisiert.

> **Profitipp**
>
> Seien Sie präzise und konsequent bei der Erstellung der UML-Diagramme! In diesem Fall können Sie erkennen, welche technischen Unterschiede aus einer anders gezeichneten Pfeilspitze resultieren können.

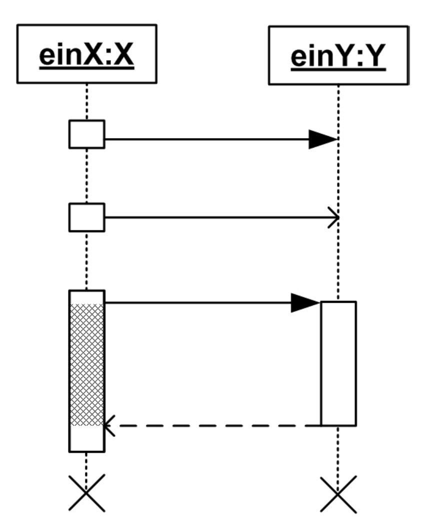


Abbildung 3.78: Synchrone und asynchrone Kommunikation

Nachdem die formale Syntax der Sequenzdiagramme nun erklärt wurde, folgen drei Beispiele aus der Anwendung dieser Diagramme.

Zunächst wird das Anlegen eines Aktiendepots und das Ausführen der ersten Order auf diesem Depot über ein Web-Frontend beschrieben. Die Aktionen gehen von einem Benutzer aus, den in diesem Fall ein Mensch darstellt. Der Akteur kann stets aktiv werden und ist nie blockiert.

Hinweis

Bitte lassen Sie einen menschlichen Akteur nicht durch ein abschließendes Kreuz an der Lebenslinie sterben. Viele Analytiker und Entwickler würden sich über Ihr UML-Diagramm lustig machen.

Im ersten Schritt klickt der angemeldete Benutzer des Bankportals auf die Schaltfläche *anlegen* und löst so einen Methodenaufruf im Web-Frontends aus. Diese Aktion legt im Backend des Portals ein neues Depot an.

Es stellt sich die Frage, wo die Berechtigung des Benutzers zum Anlegen eines neuen Depots abgeleitet werden kann. Dies kann in der Vorbedingung des zuvor entsprechend formulierten textuellen Anwendungsfalldiagramms *Depot anlegen* oder auch im entsprechenden Sequenzdiagramm eingesehen werden.

Hinweis

Auch hier erkennen Sie nochmals die Verzahnung der einzelnen Diagramme. In der Praxis ergibt sich jedoch häufig die Schwierigkeit bei größeren Projekten, diese Vielzahl von Beschreibungen konsistent zu halten. Dafür sind oft separate Mitarbeiter erforderlich.

Das „Ordern“ ist ein zweiter geschäftlicher Anwendungsfall, dessen Abbildung im technischen System durch das Sequenzdiagramm dargestellt wird. Dabei wird der entsprechende Betrag, der für den Aktienkauf verwendet werden soll, vom bereits vorhandenen Konto des Benutzers abgebucht.

Im nächsten Schritt wird die Aktienorder getätigt, bei der eine bestimmte Anzahl einer Aktienart erworben werden soll. Diese Orderanfrage wird an das neue Depotobjekt übergeben.

In dem Diagramm der Abbildung 3.79 ist nicht das komplexe System dargestellt, das sich hinter dem Depot verbirgt. Das Depot wird seinerseits eine Art von Orderobjekt anlegen und verwalten. Dieses Orderobjekt muss in Verbindung mit dem Handelsplatz einer Börse gebracht werden, wo die Order dann zur Ausführung kommt oder auch nicht. Letzteres ist beispielsweise der Fall, wenn der Benutzer einen zu geringen maximalen Kaufpreis angegeben hat.

Dieses Sequenzdiagramm beschreibt also lediglich die Aktionen, die für den Benutzer sichtbar sind und befindet sich daher auf der Wasserspiegelebene.

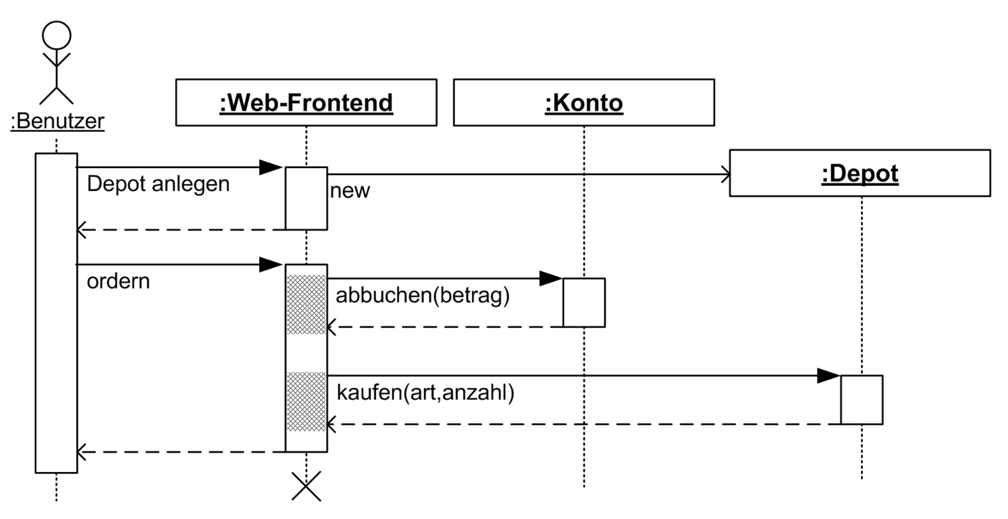


Abbildung 3.79: Anlegen eines Aktiendepots und ordern

Im nächsten Beispiel wird die Interaktion eines Apothekers mit dem technischen System einer Onlinerezeptverwaltung beschrieben. Dabei werden weitere Elemente der Syntax von Sequenzdiagrammen vorgestellt.

Der Apotheker interagiert mit einem Rezeptfenster, wobei der erste Teil der Interaktion – beispielsweise das Abzeichnen des Rezepts – nicht in diesem Diagramm dargestellt wird. Dies ist durch das Einfügen einer Interaktionsreferenz möglich, die einen Namen (hier: A) erhalten hat. Die erste Interaktion zwischen dem Apotheker und dem Rezeptfenster wird also in einem separaten Sequenzdiagramm beschrieben, welches hier nicht dargestellt wird.

Im nächsten Schritt bearbeitet der Apotheker das Rezept. Ihnen ist es vielleicht in einer Apotheke schon einmal aufgefallen, dass der Apotheker sich die erste Zeile des Rezepts

ansieht, zum Arzneischrank geht, dort das Medikament, sofern es im Lager vorhanden ist, entnimmt und Ihnen auf den Tisch legt. So geht der Apotheker Zeile für Zeile im Rezept vor. Wenn der Bestand eines Medikaments im Arzneischrank zu gering wird, bestellt der Apotheker das Medikament nach. Genau dieses Vorgehen wird im Diagramm der Abbildung 3.80 beschrieben.

Profitipp

Sie stellen sich vielleicht die Frage, warum man überhaupt so viele UML-Diagramme erstellt und nicht gleich eine präzise textuelle Beschreibung verfasst? Die Ursache liegt darin, dass diese Präzision erst durch viel Kommunikation mit Ihrem Kunden in dem iterativ-inkrementellen Prozess der Softwareentwicklung und des meist evolutionären Prototypings erreicht wird. Die UML-Diagramme eignen sich sehr gut, um den gesamten Prozess zu dokumentieren. Die Erstellung dieser Diagramme ist jedoch nicht Selbstzweck. Vielmehr sind die UML-Diagramme Hilfsmittel und Diskussionsgrundlage für die Kommunikation der Stakeholder untereinander.

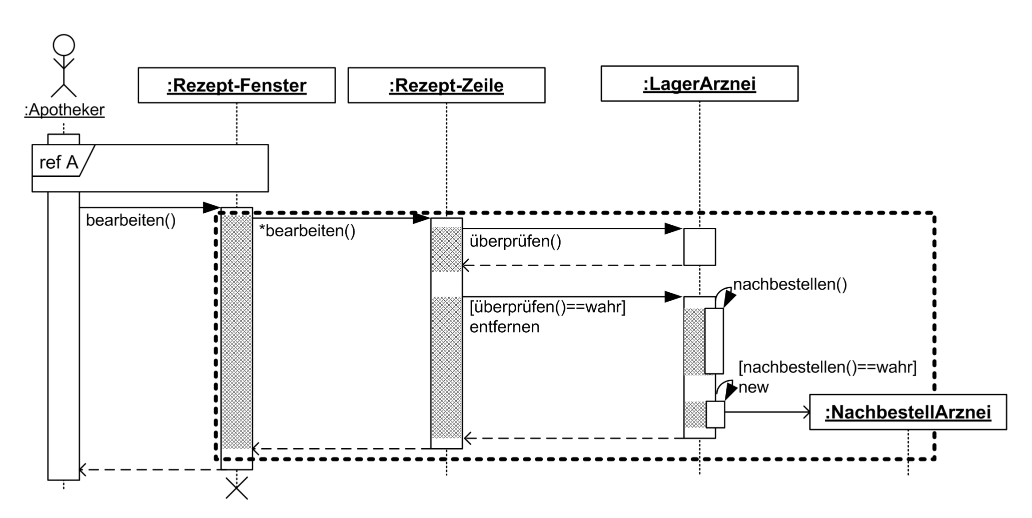


Abbildung 3.80: Abarbeitung eines Onlinerezepts

Bei dem Durchgehen der einzelnen Rezeptzeilen handelt es sich offensichtlich um eine Schleife. Deren Beginn wird in einem Sequenzdiagramm durch einen Stern dargestellt. Der Schleifenrumpf wird durch das gestrichelte Viereck dargestellt.

Sie können sich vorstellen, dass auch die Darstellung von verschachtelten Schleifen in Sequenzdiagrammen nicht sehr übersichtlich ist. Für Verzweigungen und Schleifen sind Aktivitätsdiagramme auf Fisch- und Muschelebene besser geeignet als Sequenzdiagramme. Diese wiederum haben ihre Stärke in der Darstellung der Objektinteraktion.

Eine weitere neue Notation der Abbildung 3.80 finden Sie in dem Objekt der Lagerarznei unter anderem bei dem Methodenaufruf *nachbestellen()*.

Ein Objekt kann durchaus eine eigene Methode selbst aufrufen. Dieser Selbstaufruf wird innerhalb des Aktivierungsbalkens gezeichnet und übergibt den Kontrollfluss an die aufgerufene Methode. Diese Methode kann ihrerseits wiederum Methoden auf das eigene oder auf andere Objekte aufrufen.

Das letzte Beispiel der Sequenzdiagramme beschäftigt sich wiederum mit dem Beispiel der Seminarverwaltung. In Abbildung 3.46 wurde dazu bereits ein Aktivitätsdiagramm vorgestellt.

Über das Web-Frontend gibt der Kunde mit Nachnamen *Müller* in diesem Szenario den Suchbegriff *PHP* in der Suchmaschine des Seminaranbieters ein. Die Seminarverwaltung im Backend gibt dabei alle PHP-Seminare mit den Seminarnummern und deren Termine zurück.

Herr Müller entscheidet sich für das Seminar *S1* zum Termin *T2* und möchte es jetzt buchen. Die Session des Web-Frontends merkt sich diese Auswahl.

Im nächsten Schritt gibt Herr Müller seine persönlichen Daten ein, die ebenfalls in der Session hinterlegt werden.

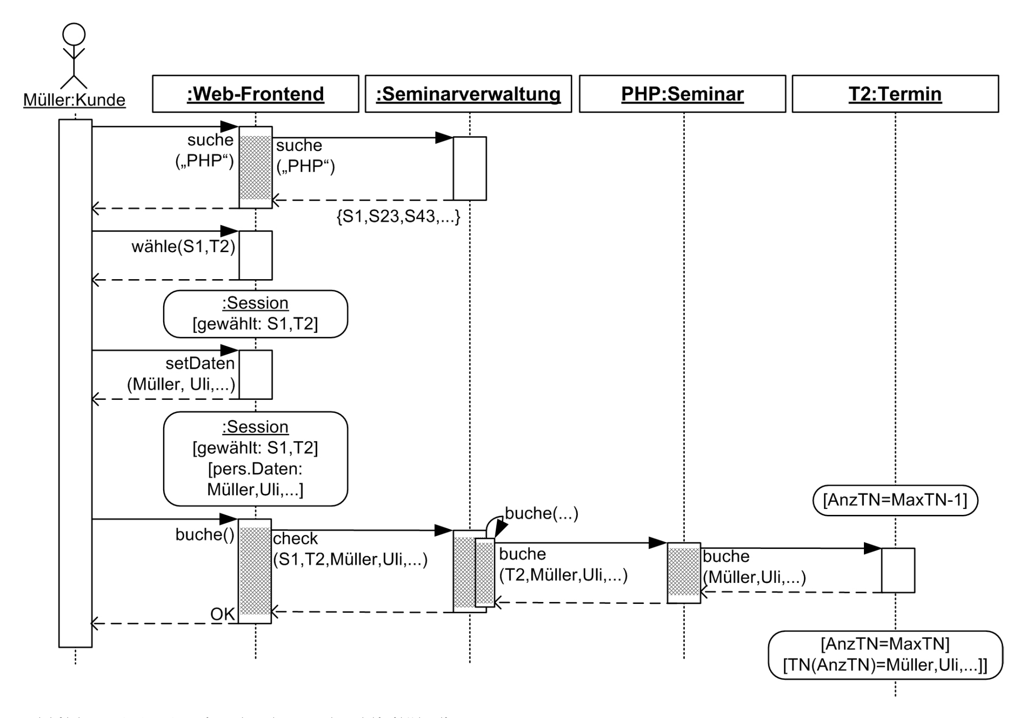


Abbildung 3.81: Seminarbuchung als Aktivitätsdiagramm

Abschließend setzt der Kunde die Buchungsanfrage verbindlich ab. Alle Daten der Session werden an die Seminarverwaltung weitergeleitet. Dort wird nochmals geprüft, ob zu diesem Termin noch ein freier Platz existiert und die Buchung über das Seminarobjekt zu dem zugeordneten Termin durchgeführt.

Herr Müller erhält abschließend eine Bestätigung, dass die Buchung erfolgreich vorgenommen wurde.

Meinung

Die eher am technischen Design orientierten Sequenzdiagramme können sehr gut mit den eher an der fachlichen Analyse ausgerichteten Aktivitätsdiagrammen abgeglichen werden. Dies gibt einen guten Anhaltspunkt, ob die fachlichen Vorgaben sinnvoll technisch umgesetzt wurden oder nicht.

Was haben Sie gelernt und wie geht es weiter?

In diesem Kapitel wurden zu Beginn verschiedene Vorgehensweisen dargestellt, wie Sie bei der Abwicklung eines Projekts vorgehen können. Es existiert keine ultimativ sinnvolle Vorgehensweise. Vielmehr ist die gewählte Vorgehensweise von der Größe der Projekte abhängig.

Die Objektorientierung eignet sich insbesondere für größere Projekte, da sonst ein „Over-Design" droht. Dazu existieren der Rational Unified Process als schwergewichtiges Modell sowie die agilen Methoden als dynamische, kommunikationsintensive Variante.

Im Anschluss daran wurden die Idee und die Grundbegriffe der Objektorientierung vorgestellt, die eine neue Denkweise bei der technischen Lösung von fachlichen Problemen implizieren.

Im letzten Teil dieses Kapitels wurden die wichtigsten Diagramme der UML vorgestellt, mit deren Hilfe Sie die Aspekte der objektorientierten Softwareentwicklung beschreiben können. Die UML-Diagramme bieten dabei eine Diskussionsgrundlage für die iterativ-inkrementelle Entwicklung und dienen zusätzlich zur Dokumentation der Entstehung einer Lösung. Die einzelnen Diagramme bieten verschiedene Sichtweisen auf das Problem und auf die Lösung. Sie geben zusätzlich durch die Reihenfolge ihrer Anwendung eine Anleitung, damit Sie sich der Problemlösung nähern können.

Alle in diesem Kapitel skizzierten Beispiele können Sie als Aufträge für die Implementierung von PHP-Anwendungen sehen, die Sie mit den neuen Möglichkeiten von PHP 5 umsetzen können.

Im nächsten Kapitel werden Sie erfahren, wie Sie die vorgestellten Konzepte der Objektorientierung und der Spezifikation der einzelnen UML-Diagramme in PHP implementieren können.

4 PHP objektorientiert

Im zweiten Kapitel dieses Buches wurde die grundlegende Syntax der Sprache PHP vorgestellt. Diese Syntax und alle vorgestellten Befehle können natürlich auch im Rahmen der Objektorientierung verwendet werden. Verzweigungen, Schleifen, Session-Handling, der Versand von E-Mails und auch die Befehle zur Datenbankanbindung stehen nach wie vor zur Verfügung. In großen Projekten sollte der gesamte Zugriff auf solche Ressourcen jedoch nur einmalig implementiert und in Objekte gekapselt werden.

Das dritte Kapitel präsentierte Vorgehensweisen bei der Abwicklung von Projekten unterschiedlicher Größe und stellte im Anschluss die grundlegenden Begriffe objektorientierter Denkweise vor. Mit der UML 2 wurde eine Sprache vorgestellt, die als Diskussionsgrundlage und zur Dokumentation von allen Beteiligten über das gesamte Projekt hinweg angewendet werden kann. Bis zu diesem Punkt ist es noch unerheblich, welche objektorientierte Sprache zur Implementierung eingesetzt wird. Anstelle von PHP können ebenso ASP.NET oder JSP/Servlets zum Einsatz kommen.

In diesem Kapitel wird nun vorgestellt, wie man die bislang theoretisch beschriebenen Konzepte der Objektorientierung mit PHP 5 umsetzen kann.

4.1 Umsetzung objektorientierter Grundlagen

Damit Sie die Umsetzung der objektorientierten Grundlagen, die in Kapitel 3.2.2 vorgestellt wurden, in PHP leicht nachvollziehen können, wird in diesem Kapitel eine Vielzahl von kleinen Beispielen vorgestellt, die jeweils einzelne Aspekte der objektorientierten Programmierung beschreiben. Im fünften Kapitel dieses Buches werden dann komplexere zusammenhängende Beispiele skizziert.

4.1.1 Die erste PHP-Klasse

Am Sinnvollsten ist es, direkt mit der Programmierung einer vollständigen Klasse zu beginnen und diese Klasse dann in einer anderen PHP-Datei mit Objekterzeugung und Ausgabe zu testen.

Eine private Eigenschaft

Die erste Klasse erzeugt einen Stift, der nur eine einzelne Eigenschaft besitzt. Er hat eine Farbe, die als Zeichenkette festgehalten wird und nicht außerhalb des Objekts zugänglich ist (*private*). Diese Datenkapselung unterscheidet die Objektorientierung von prozeduralen Ansätzen und erhöht die Wartbarkeit und Modularität.

In der Realität würde man auch eine Farbe als Objekt repräsentieren, das seinerseits drei Eigenschaften hat. Dies könnten beispielsweise die Rot-, Grün- und Blauanteile der Farbe sein. So würde man sicherstellen, dass man stets eine gültige Farbe erhält. In einer Zeichenkette könnte die Farbe auch „Frank“ lauten, was jedoch keinen Sinn macht.

Konstruktor und Destruktor

Die Klasse besitzt außerdem einen Konstruktor, der zwingend einen Parameter erhält. Dieser Parameter gibt eine Farbe von außen vor, die intern im Objekt abgelegt wird. Ein Default-Konstruktor ist in diesem Beispiel nicht vorgesehen. Sie müssen bei der Objekterzeugung also den Farbparameter angeben. In PHP 5 lauten alle Methoden, die Konstruktoren für ein Objekt sind, *__construct*. Sie können diese Methode mehrmals mit einer unterschiedlichen Anzahl an Parametern verwenden, sodass Sie mehrere mögliche Konstruktoren für ein Objekt definieren können.

Zusätzlich besitzt die Klasse einen Destruktor, der beim Entfernen des Objekts aus dem Arbeitsspeicher des Servers automatisch aufgerufen wird. Ein Destruktor wird über den Aufruf *__destruct* definiert und ist stets parameterlos.

Get- und Set-Methoden

Zusätzlich erhält ein Stift zwei Methoden, die als Dienste von anderen Klassen aus angesprochen werden können (*public*). Die erste Methode kann dem Stift eine neue Farbe geben; es handelt sich um einen so genannten „Setter“, der eine Eigenschaft neu setzt. Ob das Setzen des übergebenen Parameters erlaubt ist, kann die Klasse selbst in der Programmierung der Set-Methode entscheiden. Damit wird die Datenkapselung der Objektorientierung gewährleistet. Die zweite Methode gibt den Wert der Eigenschaft zurück. Es handelt sich also um eine Get-Methode.

Abbildung 4.1 skizziert zunächst auf der linken Seite ein Stiftobjekt und auf der rechten Seite das UML Klassendiagramm der Designphase.

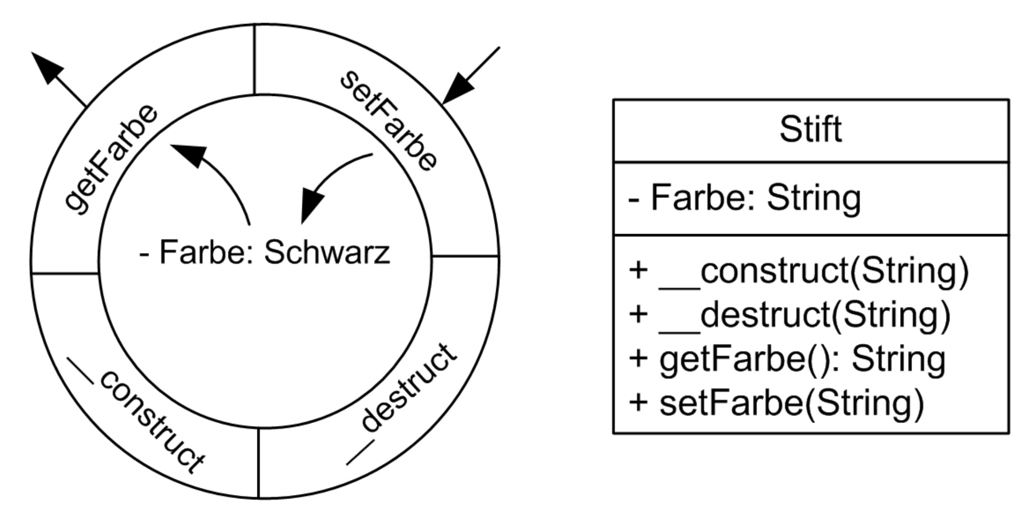


Abbildung 4.1: Das erste zu erstellende Objekt und das passende Klassendiagramm

Dieses Klassendiagramm wird nun direkt in PHP-Code umgesetzt. Die UML-Klassendiagramme in der Designphase sollten so präzise sein, dass Sie keine Möglichkeit zur Interpretation mehr zulassen.

Meinung

Aus Erfahrung und zur Vergleichbarkeit mit anderen objektorientierten Sprachen sollte jede Klasse in eine separate Datei geschrieben werden. Diese Datei sollte genauso heißen wie der Name der Klasse.

Da es keinen Sinn macht, eine Klassendefinition direkt zu verwenden und diese stets von anderen PHP-Dateien eingebunden wird, sollte die Datei *stift.inc.php* genannt werden.

```
<?php
class Stift{
  private $farbe; // Eigenschaft des Stiftes

  // Konstruktor
  public function __construct($farbe){
    echo('Ein Stift der Farbe '.$farbe.' wird erzeugt...<br>');
    $this->setFarbe($farbe);
  }

  // Destruktor
  public function __destruct(){
  echo('Der Stift der Farbe '.$this->farbe.' ist jetzt zerstört.<br>');
  }

  public function getFarbe(){
    return $this->farbe;
  }

  public function setFarbe($farbe){
    $this->farbe=$farbe;
  }
}
?>
```

Listing 4.1: Die erste Klasse stift.inc.php

Was ist $this?

Bei allen Methoden fällt eine Variable *$this* auf, die in dieser Klasse nie definiert wird. Die so genannte *$this*-Referenz existiert bei jedem Objekt und ist ein Zeiger auf sich selbst. Da wir uns in der Klasse *Stift* befinden, können Sie über den Befehl *$this->setFarbe($farbe)* im

Konstruktor auf dem Objekt selbst einen Methodenaufruf tätigen. Dieser Selbstaufruf wurde in den Sequenzdiagrammen (Kap. 3.2.6; Abläufe im technischen Modell: Sequenzdiagramme) beschrieben.

Im zweiten Kapitel haben Sie bereits erfahren, dass Sie Variablen in PHP nicht vor der Verwendung deklarieren müssen. Bei der prozeduralen Programmierung ist dies ein angenehmes Verhalten, das Codezeilen einspart.

Bei der Methode *setFarbe* führt sie jedoch zu einem Problem. Sie müssen hier die Eigenschaft des Objekts, die privat deklariert ist, mit einem von außen übergebenen Parameter *$farbe* überschreiben. PHP kann aber eine in der Methode lokal definierte Variable erst einmal nicht von dem Zugriff auf eine deklarierte Eigenschaft des Objekts unterscheiden. Wenn Sie auf die Eigenschaft zugreifen wollen, müssen Sie daher auch die *$this*-Referenz verwenden, indem Sie mit dem Befehl *$this->farbe=$farbe* der Eigenschaft des Objekts den von außen übergebenen Wert zuweisen. Der Zugriff auf die private Eigenschaft ist an dieser Stelle möglich, da Sie sich ja in der Klasse selbst befinden.

Die Objekterzeugung

Nachdem die Klasse als Bauplan für Objekte erstellt wurde, können Sie nun versuchen, ein erstes Objekt dieser Klasse anzulegen. Dies funktioniert sehr leicht, indem Sie im ersten Schritt eine zusätzliche PHP-Datei erstellen und die Klassendatei über *require_once* einbinden. Eine mögliche mehrfache Einbindung würde zu einer mehrfachen Klassendeklaration mit demselben Namen und damit zu einem Fehler führen.

Im HTML-Rumpf der PHP-Datei kann nun das erste Objekt angelegt werden. Für die Erzeugung von neuen Objekten bietet PHP, wie auch die meisten anderen objektorientierten Sprachen, das Schlüsselwort *new* an. Hinter diesem Schlüsselwort müssen Sie den Namen der Klasse angeben, von der Sie ein neues Objekt anlegen wollen. Wenn Sie keine Parameter angeben, wird der parameterlose Default-Konstruktor der Klasse aufgerufen. Dieser existiert jedoch bei unserer Klasse nicht. Stattdessen müssen Sie eine Zeichenkette als Parameter übergeben, der die Farbe des zu erstellenden Stiftes darstellt. In diesem Fall wird ein schwarzer Stift erzeugt.

Der *new*-Operator gibt als Ergebnis eine Referenz auf das gerade erzeugte Objekt zurück, die man wiederum in einer Variablen abspeichern kann. Diese Variable heißt hier *$einStift*.

```
<?php require_once("stift.inc.php"); ?>
<html><body>
  <?php
    // kein Default-Konstruktor erlaubt
    // $einStift=new Stift();
    $einStift=new Stift('schwarz');
    echo('Farbe: '.$einStift->getFarbe().'<br>');
    $einStift->setFarbe('blau');
    echo('Farbe: '.$einStift->getFarbe().'<br>');
```

Listing 4.2: Die erste Objekterzeugung – ein Stift wird geboren

```
    var_dump($einStift);echo('<br>');
  ?>
</body></html>
```

Listing 4.2: Die erste Objekterzeugung – ein Stift wird geboren (Forts.)

Das gerade erzeugte Objekt hat also seinen Konstruktoraufruf erfolgreich abgeschossen und besitzt nun eine Farbe. Da die PHP-Datei aus Listing 4.2 eine Referenz auf dieses Objekt besitzt, kann man von hier aus auf alle öffentlich zugänglichen Methoden zugreifen. Sie können das Objekt mit dem Befehl *$einStift->getFarbe()* fragen, welche Farbe es besitzt. Das Ergebnis wird dann ausgegeben.

In der nächsten Zeile bekommt der Stift dann über *$einStift->setFarbe('blau')* eine neue Farbe zugewiesen. Auch diese neue Farbe können Sie wieder abfragen.

Interessant ist auch, wenn Sie mithilfe des aus dem zweiten Kapitel bekannten Befehls *var_dump* das Objekt ausgeben. Sie erhalten dann eine detaillierte Ausgabe der internen Eigenschaften und deren aktueller Ausprägung. Dies entspricht genau den Daten eines Objektdiagramms. Die erste Zeile ist die Ausgabe des Konstruktors; in der letzten Zeile wird der Stift durch den Destruktor-Aufruf zerstört. Das Objekt „lebt“ also nur während des Aufrufs der PHP-Datei aus Listing 4.2. Die Ausgabe des Listings lautet

Ein Stift der Farbe schwarz wird erzeugt...

Farbe: schwarz

Farbe: blau

object(Stift)#1 (1) { ["farbe:private"]=> string(4) "blau" }

Der Stift der Farbe blau ist jetzt zerstört.

4.1.2 Objekte in einer Session übergeben

Die ganze Theorie der Objektorientierung und der Aufwand der Implementierung würden sich nicht lohnen, wenn die Objekte und deren Beziehungen nur während eines PHP-Aufrufs bestehen würden. Sie müssen also dafür sorgen, dass die Objekte über einen längeren Zeitraum bestehen können. Das erste Ziel besteht darin, Objekte in einer Session abzulegen, um die Referenzen im nächsten Aufruf einer PHP-Seite weiterverwenden zu können.

Um das Speichern eines Objekts in einer Session zu testen, muss der Anwender zunächst in einem Eingabeformular die Farbe eines Stifts in einem Textfeld eingeben. In diesem Formular wird auch die Session gestartet. Der Quellcode des Formulars ist in Listing 4.3 dargestellt.

```
<?php session_start(); ?>
<html><body>
  <h1>Willkommen in der Stift-Herstellung</h1>
```

Listing 4.3: Quellcode des Eingabeformulars

```
  <form action="stifterzeugung.php" method="post"><pre>
    Farbe: <input name="frmFarbe" type="text"><br>
    <input value="Herstellen..." type="submit"><br>
    Sie haben die Session-ID <?php echo session_id()?> vom Server erhalten.
  </pre></form>
</body></html>
```

Listing 4.3: Quellcode des Eingabeformulars (Forts.)

In der Datei, die über HTTP-Post aufgerufen wird, wird nun der Stift erzeugt und in die Session übergeben. Der Quellcode der Datei ist in Listing 4.4 aufgeführt.

```
<?php
  session_start();
  require_once("stift.inc.php");
  $frmFarbe=$_POST[frmFarbe];
  $einStift=new Stift($frmFarbe);
  $_SESSION[StiftContainer]=$einStift;
?>
<html><body>
  Ein Stift der Farbe <?php echo $einStift->getFarbe()?> wurde erzeugt.<br>
  <a href="weiter.php">Weiter gehts...</a><br>
</body></html>
```

Listing 4.4: Objekterzeugung und Speicherung in der Session

Die Ausgabe des Listings lautet überraschenderweise

Ein Stift der Farbe rot wird erzeugt...

Ein Stift der Farbe rot wurde erzeugt.

Weiter gehts...

Der Stift der Farbe rot ist jetzt zerstört.

Es stellt sich die Frage, warum der Destruktor des Stifts aufgerufen wird, obwohl das Stiftobjekt doch in der Session persistent gehalten werden soll? Ist das Stiftobjekt in der Session jetzt verloren?

Die Antwort lautet: Nein, das Stiftobjekt in der Session ist noch vorhanden! Die Sessiondatei befindet sich auf dem Dateisystem des PHP-Servers. In diese Datei wurde das Objekt „hineinserialisiert“ – siehe dazu auch das nächste Kapitel zur Serialisierung. Lediglich das Stiftobjekt aus dem Arbeitsspeicher des PHP-Servers wurde wieder freigegeben.

Die Daten der Session können Sie sich übrigens ansehen. Wenn Sie das XAMPP-Paket installiert haben, wurde in dessen Unterverzeichnis *tmp* mit dem ersten Start der Session eine Datei angelegt. Falls die Session mit der ID *3217d032e41dfe5fa52a481b35391302* angelegt wurde, so heißt die Datei *sess_3217d032e41dfe5fa52a481b35391302*. Diese Datei kön-

nen Sie mit einem Texteditor öffnen und finden das serialisierte Objekt mit seiner Eigenschaft: *StiftContainer|O:5:"Stift":1:{s:12:" Stift farbe";s:3:"rot";}*

Das O bedeutet, dass im Folgenden ein Objekt abgespeichert ist. Dann kommt die Anzahl der Zeichen für den Klassennamen. Die 1 ist die erste Eigenschaft, die aus einer Zeichenkette (*s* für String) besteht. Dem folgt der Name der Eigenschaft in 12 Zeichen, wobei der Name aus dem Namen der Klasse und dem Namen der eigentlichen Eigenschaft zusammengesetzt ist. Der Wert der Eigenschaft folgt dann hinter dem Semikolon. Es ist eine Zeichenkette mit drei Zeichen und der Wert lautet *rot*.

Mit einem Klick auf den Link der *weiter.php* sehen Sie nun, dass man immer noch auf das entstandene Objekt zugreifen kann. Listing 4.5 zeigt den Quellcode dieser Datei.

```
<?php
  session_start();
  require_once("stift.inc.php");
  $einContainerStift=$_SESSION[StiftContainer];
?>
<html><body>
  Sie besitzen einen Stift der Farbe
  <?php echo $einContainerStift->getFarbe()?>.<br>
</body></html>
```

Listing 4.5: Objekterzeugung und Speicherung in der Session

In dieser Datei wird zunächst wieder die Klassendefinition eingebunden. Im Anschluss daran wird die Session wieder gestartet und das Stiftobjekt aus der Session heraus in den Arbeitsspeicher kopiert. Man spricht in diesem Zusammenhang von einer Deserialisierung. Im Folgenden sehen Sie die Ausgabe der PHP-Datei:

Sie besitzen einen Stift der Farbe rot.

Der Stift der Farbe rot ist jetzt zerstört.

Da der Stift bereits existierte und nur noch vom Dateisystem in den Arbeitsspeicher geladen wird, erfolgt kein erneuter Aufruf des Konstruktors. Die zweite Ausgabe zeigt wiederum den Aufruf des Destruktors des Objekts im Arbeitsspeicher des Servers. Die serialisierte Kopie existiert jedoch solange, wie die Session existiert. Beim Löschen der Session wird jedoch der Destruktor nicht aufgerufen, da das Objekt im Dateisystem lediglich in Textform vorliegt und nicht aktiv ist.

Hinweis

In der Realität werden natürlich keine Stifte gespeichert, sondern Warenkörbe oder Objekte in einer Datenbank. Die Erzeugung von einfachen Objekten ist aber für das Erlernen der Objektorientierung sehr hilfreich, um deren Prinzipien kennen zu lernen.

4.1.3 Objekte speichern und laden: (De-)Serialisierung

Die im vorherigen Kapitel besprochene Serialisierung in einer Session kann auch manuell durch den Befehl *serialize* angestoßen werden. Sie erhalten dadurch ein serialisiertes Objekt in einer Zeichenkette. Dieses Objekt können Sie dann in einem versteckten Textfeld über mehrere Formulare übertragen oder auch in einer Datei ablegen.

Der Quellcode aus Listing 4.6 zeigt in einer veränderten *stifterzeugung.php* das Serialisieren des Stiftobjekts und das Festhalten der serialisierten Daten in der Variable *$ser*. Die serialisierte Zeichenkette des Objekts wird dann in die Datei *objekt.txt* abgelegt.

```
<?php
  require_once("stift.inc.php");
  $frmFarbe=$_POST[frmFarbe];
  $einStift=new Stift($frmFarbe);
  $ser=serialize($einStift);
  $datei=@fopen("objekt.txt","w");
  @fwrite($datei, $ser);
  @fclose($datei);
?>
<html><body>
  Ein Stift der Farbe <?php echo $einStift->getFarbe()?> wurde erzeugt.<br>
  <a href="weiter.php">Weiter gehts...</a><br>
</body></html>
```

Listing 4.6: Speicherung eines serialisierten Objekts in einer Datei

Diese Datei wird in Listing 4.7 in der veränderten *weiter.php* wieder ausgelesen, das Objekt mit dem Befehl *unserialize* wieder hergestellt und eine Methode des Objekts aufgerufen.

Die Daten des Objekts werden in der *objekt.txt* in der gleichen Form gespeichert wie in der Session. Beim Speichern in eine Session und dem Laden aus einer Session wird also automatisch eine Serialisierung bzw. Deserialisierung durchgeführt.

```
<?php
  require_once("stift.inc.php");
  $datei=@fopen("objekt.txt","r");
  $deser = unserialize(@fgets($datei));
  @fclose($datei);
?>
<html><body>
  Sie besitzen einen Stift der Farbe <?php echo $deser->getFarbe()?>.<br>
</body></html>
```

Listing 4.7: Laden und Deserialisieren des Objekts aus der Datei

4.1.4 PHP-eigene Methoden der Objektorientierung

Auf Grund der historischen Entwicklung der Sprache PHP von den „Personal Homepage Tools“ hin zu einer objektorientierten Programmiersprache sind einige besondere Funktionen entstanden, die für andere objektorientierte Programmiersprachen eher unüblich sind.

Diese Funktionen werden verstärkt in diesem Kapitel behandelt und ihre Bedeutung für die Umsetzung der Konzepte der Objektorientierung wird herausgestellt.

Automatisches Nachladen von Klassen

Wenn Sie ein Objekt einer Klasse erzeugen wollen, muss der PHP-Datei, in der das Objekt erzeugt werden soll, die Definition der Klasse bekannt sein. Nach diesem Bauplan wird das Objekt dann angelegt. So sehen Sie unter anderem in Listing 4.7 des letzten Kapitels die Einbindung der Klassenbeschreibung für Stifte durch den Befehl *require_once("stift.inc.php")*.

Da die Objektorientierung sich gerade für große Projekte mit komplexen Objektgeflechten eignet, ist diese Deklaration jeder einzelnen Klasse zu Beginn jeder PHP-Datei sehr wartungsaufwändig.

Um diesen Aufwand zu minimieren, bietet PHP die *__autoload*-Methode. Wie der Konstruktor und der Destruktor sind die Spezialmethoden der Objektorientierung bei PHP durch den Beginn mit zwei Unterstrichen gekennzeichnet.

Die *__autoload*-Methode wird dann aufgerufen, wenn ein Objekt einer Klasse erzeugt werden soll, die Definition der Klasse aber noch nicht bekannt ist. Wenn Sie diese Methode einmalig programmieren, können Sie das Einbinden von Klassendefinitionen durch den Aufruf von *require_once* automatisieren. Dabei können Sie auch Ihren Standardpfad angeben, in dem Sie alle Klassendefinitionen verwalten.

So zeigt Listing 4.8 einen einfachen Klassenlader, der beim Fehlen der im Eingabeparameter übergebenen Klasse *$klasse* automatisch die entsprechende *.inc.php*-Datei nachlädt, die sich in diesem Fall im selben Verzeichnis befinden muss.

```
<?php
  function __autoload($klasse){
    require_once($klasse.".inc.php");
  }
?>
```

Listing 4.8: Ein einfacher Klassenlader classloader.inc.php

Diesen kleinen Klassenlader können Sie nun in jede PHP-Datei einbauen, die Objekte Ihrer eigenen Klassen anlegt. Listing 4.9 zeigt die Anwendung des Klassenladers auf das bereits beschriebene Listing 4.6, bei dem ein Stiftobjekt angelegt wird.

Wenn Sie statt eines Objekts 50 Objekte unterschiedlicher Klassen anlegen wollen, werden Sie das automatische Laden der Klassen zu schätzen wissen und 49 Zeilen PHP-Code einsparen.

```
<?php
  require_once("classloader.inc.php");
  $frmFarbe=$_POST[frmFarbe];
  $einStift=new Stift($frmFarbe);
  $ser=serialize($einStift);
  $datei=@fopen("objekt.txt","w");
  @fwrite($datei, $ser);
  @fclose($datei);
?>
<html><body>
  Ein Stift der Farbe <?php echo $einStift->getFarbe()?> wurde erzeugt.<br>
  <a href="weiter.php">Weiter gehts...</a><br>
</body></html>
```

Listing 4.9: Anwendung des Klassenladers

Zentrale Verwaltung von Get- und Set-Methoden

In der Definition der Stiftklasse haben Sie bereits gesehen, dass die einzige Eigenschaft *$farbe* eine Get- und eine Set-Methode zur Verwaltung des Zugriffs auf diese Eigenschaft besitzt.

Eine ähnliche Eigenschaft besitzt die Klassendefinition *Laender*, bei der intern ein Datenfeld *$laender* mit Namen und Kürzel von zwei Ländern verwaltet wird. Die Notation => legt dabei ein assoziatives Feld an (Listing 2.13).

PHP bietet Ihnen im Gegensatz zu Java die Möglichkeit an, die Get- und Set-Methoden zentral zu verwalten. So erlaubt die *__get*-Methode in Listing 4.10 den Aufruf von *$land1=$x->Deutschland* bei einem Objekt *$x* der *Laender*-Klasse. Da die Eigenschaft privat deklariert ist, ist dieser Zugriff zunächst verboten. Über die *__get*-Methode können Sie einen solchen Zugriff jedoch innerhalb der Klasse definieren. So gibt die *__get*-Methode das entsprechende Länderkürzel aus dem assoziativen Datenfeld zurück.

Wie bei der herkömmlichen Set-Methode der Stiftklasse kann die allgemeine *__set*-Methode Elemente aus dem existierenden Datenfeld verändern, indem beim ersten Eingabeparameter das Land und im zweiten Eingabeparameter das neue Länderkürzel übergeben wird.

```
<?php
  class Laender{
    private $laender=array("Deutschland"=>"D","England"=>"EN");

    public function __get($var){
      return $this->laender[$var];
    }

```

Listing 4.10: Länderkürzel und Get-/Set-Verwaltung

```
    public function __set($var,$wert){
      $this->laender[$var]=$wert;
    }
  }
?>
```

Listing 4.10: Länderkürzel und Get-/Set-Verwaltung (Forts.)

In Listing 4.11 wird nun ein neues Länderobjekt angelegt. Es sieht so aus, als könnten Sie im Aufruf *$x->Deutschland="DE"* den Wert einer öffentlich zugänglichen Eigenschaft mit dem Namen *Deutschland* direkt überschreiben. Dies würde der Datenkapselung der Objektorientierung widersprechen. In Wirklichkeit wird innerhalb der Klasse jedoch die *__set*-Methode aufgerufen. Dort können durchaus Überprüfungen der Eingabe, in diesem Fall der Eingabe *DE* stattfinden. Die Klasse behält also die Hoheit darüber, wie ihre internen Eigenschaften gesetzt werden. Dies entspricht der objektorientierten Denkweise. Ein Anwender der Klasse merkt nicht einmal, dass die Länderkürzel innerhalb der Klasse in einem assoziativen Feld verwaltet werden.

```
<?php
  require_once("classloader.inc.php");
  $x=new Laender();
  $x->Deutschland="DE";
  $land1=$x->Deutschland;
  $x->Frankreich="FR";
  $land2=$x->Frankreich;
?>
<html><body>
  <?php echo $land1?><br> <?php echo $land2?><br>
</body></html>
```

Listing 4.11: Anwendung der Länderklasse

In der gleichen Weise wird durch den Befehl *$land1=$x->Deutschland* in Wirklichkeit die *__get*-Methode aufgerufen, die auf das assoziative Feld zugreift und den neuen, im Objekt hinterlegten Ländercode für Deutschland ausliest.

Interessant ist auch der Aufruf *$x->Frankreich="FR"*, denn die Eigenschaft *Frankreich* existiert noch nicht einmal im internen Datenfeld *$laender* der Klasse. Mit der *__set*-Methode sind Sie also in der Lage, eine Klasse zur Laufzeit nach außen hin dynamisch um neue Eigenschaften zu erweitern. In Wirklichkeit wird dem assoziativen Feld lediglich ein neuer Eintrag hinzugefügt. Bei einer Serialisierung des Objekts wird das Feld mit serialisiert und die neue „Eigenschaft“ gespeichert. Das Feld kann natürlich auch bei einer Speicherung in einer Datenbank mit seinem gesamten Inhalt persistent, also dauerhaft, gespeichert werden.

Aufrufen nichtdeklarierter Methoden und Identifikation von Objekten

Ähnlich wie der Zugriff auf Eigenschaften können sogar Methodenaufrufe von einer besonderen PHP-Funktion manipuliert werden. Diese Funktion nennt sich *__call*. Sie ist insbesondere dafür geeignet, einen Nachteil der Sprache PHP in der Objektorientierung auszugleichen.

Wie Sie wissen, müssen in PHP Variablen nicht mit Datentypen deklariert werden. Sie sagen also nicht vor der Verwendung von *$x*, dass *$x* eine Ganzzahl oder eine Zeichenkette sein muss. PHP verwaltet die Datentypen bei der Zuweisung eines Wertes dynamisch. So kann sich der Datentyp einer Variablen sogar während ihrer Lebenszeit ändern. In der Praxis des prozeduralen Programmierens wird diese Fähigkeit des PHP-Interpreters gerade von Programmieranfängern geschätzt. Man muss sich keine Gedanken um Datentypen machen.

In der Objektorientierung ist diese Fähigkeit der Sprache jedoch erst einmal nachteilig. In Abbildung 3.65 wurde bereits ein Klassendiagramm einer Bruchklasse dargestellt, das an dieser Stelle nochmals in Abbildung 4.2 aufgegriffen wird.

Bruch
- zähler: int = 0 - nenner: int = 1 - anzahl: int = 0
+ Bruch(int,int) + add(int) + add(double) + add(Bruch) + isNull(): boolean + getAnzahl(): int

Abbildung 4.2: UML-Klassendiagramm der Bruchklasse

Dieses Klassendiagramm ist zunächst unabhängig von einer Programmiersprache. Wir konzentrieren uns zunächst auf die Eigenschaften *Zähler* und *Nenner* sowie auf den Konstruktor und auf die Methoden der Addition einer ganzen Zahl und eines anderen Bruchs. Die Prüfung der Gültigkeit eines Bruchs, die ja nur bei einem Nenner ungleich 0 gegeben ist, wird zunächst zurückgestellt. Die erste Realisierung der Bruchklasse sehen Sie in Listing 4.12.

Die Vorbelegung des Zählers und Nenners mit Standardwerten kann PHP leisten, indem die Variablen vorbelegt werden. Damit erhalten Sie auch gleichzeitig den Integer-Datentyp. Zusätzlich werden für den Zähler und den Nenner passende Get- und Set-Methoden definiert. In der Set-Methode des Nenners kann später die Prüfung der Eingabe erfolgen, die ungleich 0 sein muss. Der Konstruktor des Bruchs ruft dann die Set-Methoden auf und belegt die Eigenschaften des neuen Objekts mit den übergebenen Werten.

Problematisch an der Vorgabe aus dem Klassendiagramm ist die Existenz mehrerer *add*-Methoden, die sich lediglich anhand des Datentyps unterscheiden. In der Deklaration einer PHP-Funktion können Sie jedoch keinen Datentyp angeben, wie dies bei Java oder VB.NET möglich ist. Dort würden Sie mehrere *add*-Methoden schreiben, die sich nicht im Namen, jedoch am Datentyp des übergebenen Eingabeparameters unterscheiden. Der Java- oder .NET-Interpreter würde dann die passende Methode auswählen.

Genau diesen Schritt nimmt Ihnen PHP nicht ab. Sie müssen die Auswahl der richtigen Funktion selbst implementieren. Es wird entweder eine ganze Zahl, oder ein anderer Bruch übergeben. An dieser Stelle kommt die *__call*-Funktion ins Spiel.

Wenn Sie diese besondere PHP-Funktion in einer Klasse implementieren, wird sie aufgerufen, wenn Sie eine Methode eines Objekts aufrufen, die gar nicht existiert. Den Aufruf können Sie dann in der *__call*-Funktion umleiten.

```
<?php
  class Bruch{
    private $z=0; private $n=1;

    public function __construct($z,$n){
      $this->setZähler($z); $this->setNenner($n);
    }

    public function getZähler(){
      return $this->z;
    }
    public function getNenner(){
      return $this->n;
    }

    public function setZähler($z){
       $this->z=$z;
    }
    public function setNenner($n){
      $this->n=$n;
    }

    public function ausgeben(){
      return $this->getZähler()."/".$this->getNenner();
    }

    public function __call($func,$data){
      if ($func=="add"){
```

Listing 4.12: Die erste Bruchklasse

```
        if (is_integer($data[0])){
          $this->add_int($data[0]);
        }
        else{
          if ($data[0] instanceof Bruch){
            $this->add_bruch($data[0]);
          }
        }
      }
    }

    private function add_int($int){
      $this->z += $int * $this->n;
    }
    private function add_bruch($b){
      $this->z = $this->z * $b->getNenner() + $this->n * $b->getZähler();
      $this->n *= $b->getNenner();
    }
  }
?>
```

Listing 4.12: Die erste Bruchklasse (Forts.)

Die *__call*-Funktion besitzt zwei Eingabeparameter. Im ersten Parameter wird der Name der aufgerufenen Methode übergeben. Der zweite Parameter enthält eine Liste der übergebenen Parameter an diese aufgerufene Funktion. Die Abfrage in der Klasse prüft zuerst, ob der Name der Methode *add* lautet.

In diesem Fall wird der erste Parameter, der in *$data[0]* enthalten ist, auf seinen Datentyp geprüft. Ist es eine Ganzzahl, so wird die private Hilfsmethode *add_int* aufgerufen und der übergebene Parameter an diese Methode weitergereicht.

Die Klassenzugehörigkeit eines Objekts können Sie mit dem Befehl *instanceof* prüfen. Achten Sie hier auf die Syntax: Der Name der abgefragten Klasse wird **nicht** in Klammern gesetzt! Wenn der übergebene Parameter ein Objekt der Klasse *Bruch* ist, so ist die Bedingung der zweiten *if*-Verzweigung wahr. In diesem Fall wird die privat deklarierte Hilfsmethode *add_bruch* aufgerufen und die Objektreferenz weitergegeben.

Die Hilfsmethoden addieren nun gemäß den Regeln der Bruchrechnung die ganze Zahl bzw. den anderen Bruch zum eigenen Bruchobjekt.

Um den aktuellen Wert des Bruchs auszugeben, wurde zusätzlich die Methode *ausgeben* definiert, die den Zähler und Nenner durch einen Schrägstrich getrennt als Zeichenkette zurückgibt. Eine elegantere Methode dazu finden Sie im nächsten Unterkapitel.

Interessant ist die Frage, wie der Aufruf einer solchen *add*-Methode über die *__call*-Konstruktion funktioniert. Dies ist in Listing 4.13 dargestellt. Zunächst werden zwei neue Brüche erzeugt und der erste Bruch 1/2 ausgegeben.

Überraschenderweise sieht die Verwendung der add-Methode in *$a->add(3)* so aus, als würde die Bruchklasse eine Methode *add(int)* besitzen. Genau dies fordert ja das UML-Diagramm der Abbildung 4.2 aus der Spezifikation. Die interne Realisierung über die *__call*-Konstruktion ist also nach außen nicht sichtbar.

Im Anschluss an die Addition der ganzen Zahl erfolgt zur Kontrolle eine neue Ausgabe. Der Bruch hat nun korrekterweise den Wert 7/2.

```
<?php require_once("classloader.inc.php"); ?>
<html><body>
<?php
  $a=new Bruch(1,2);
  $b=new Bruch(3,2);
  echo $a->ausgeben().'<br>';
  $a->add(3);
  echo $a->ausgeben().'<br>';
  $a->add($b);
  echo $a->ausgeben().'<br>';
?>
</body></html>
```

Listing 4.13: Ein Beispiel der Bruchrechnung

Abschließend wird in der gleichen Art und Weise der Bruch *$b* zum aktuellen Stand des Bruchs *$a* addiert. Dies erfolgt quasi über den Methodenaufruf *add(Bruch)*. Die abschließende Ausgabe zeigt auch hier das korrekte Ergebnis; der Bruch *$a* hat jetzt den Wert 20/4.

Umwandeln von Objekten in Zeichenketten

Der Name der Methode *ausgeben()* zur Rückgabe des aktuellen Wertes eines Bruchs ist frei gewählt. Man hätte den Namen auch *print()* oder *getDaten()* nennen können. Wenn jeder Programmierer eigene Namen zur Ausgabe des Objekts erfinden würde, gäbe es keinen Standard und man müsste stets in der Dokumentation nachsehen.

Ein Standard besteht darin, zu Debug-Zwecken den aktuellen Stand aller Eigenschaften eines Objekts auszugeben. So ergibt die Ausgabe *var_dump($a);* des zuvor definierten Bruchs *$a=new Bruch(1,2);* die folgende Ausgabe: *object(Bruch)#1 (2) { ["z:private"]=> int(1) ["n:private"]=> int(2) }*.

Für einen Programmierer sind diese Daten sehr hilfreich, der Anwender würde von einer solchen Ausgabe jedoch eher abgeschreckt. Was geschieht eigentlich, wenn Sie einfach das Objekt mittels *echo $a;* im PHP-Code ausgeben? Eine Zahl können Sie ja auch auf diese Art und Weise ausgeben. Wenn Sie eine solche Ausgabe bei einem Objekt versuchen, liefert PHP seit der Version 5.2 jedoch die folgende Ausgabe:

Catchable fatal error: Object of class Bruch could not be converted to string in C:\EigeneDateien\HTTP\klassen1c\bruchrechnen2.php on line 7

Damit Sie ein Objekt in eine Zeichenkette konvertieren können, müssen Sie eine weitere besondere PHP-Methode in Ihrer Klasse implementieren. Die ursprüngliche Methode *ausgeben()* wird umbenannt in *__toString()* und sieht nun folgendermaßen aus:

```
public function __toString(){
  return $this->getZähler()."/".$this->getNenner();
}
```

Listing 4.14: Implementierte toString-Methode der Bruchklasse

Nach der Implementierung der Methode in der Klasse können Sie *echo $a;* fehlerfrei ausführen und erhalten die Ausgabe 1/2 als Zeichenkette.

4.1.5 Einzigartige Eigenschaften und Methoden

Bereits in der Beschreibung objektorientierter Modellierung wurden in Kapitel 3.2.2 Klassenattribute und Klassenmethoden erwähnt. Im Gegensatz zu herkömmliche Eigenschaften wie dem Zähler oder Nenner eines Bruchs existieren Klassenattribute nur einmal pro Klasse und gehören zur Klasse selbst, also nicht zu jedem Objekt.

Man benötigt Klassenattribute also, um Eigenschaften einer Klasse selbst abzuspeichern. Das sind meist statistische Informationen, wie die Anzahl erzeugter Objekte, der Tariflohn einer Tarifklasse oder der minimale und maximale Lohn einer Klasse von Mitarbeitern.

Auch in der Bruchklasse ist ein Klassenattribut vorgesehen, das in dem UML-Klassendiagramm von Abbildung 4.2 unterstrichen dargestellt wird und die Anzahl der erzeugten Brüche verwaltet. Ebenso wie gewöhnliche Eigenschaften sollten Klassenattribute als *private* deklariert sein und über Methodenaufrufe zugänglich gemacht werden. Diese Methoden gehören dann ebenfalls zur Klasse selbst und werden Klassenmethoden genannt. In Abbildung 4.2 ist eine Get-Methode vorgesehen.

Listing 4.15 zeigt die Erweiterung der Bruchklasse um das Klassenattribut *$anzahl* und um die Klassenmethode *getAnzahl()*. Gehören eine Eigenschaft oder eine Methode zu einer Klasse, so geschieht dies über das Schlüsselwort *static*.

Eine Klassenmethode kann auch dann aufgerufen werden, wenn noch gar kein Objekt der Klasse existiert; die Methode gehört schließlich zur Klasse selbst. Dies ist gerade bei der Anfrage nach der Anzahl der erstellten Objekte sinnvoll. Immer, wenn ein neues Objekt angelegt wird, wird der Zähler inkrementiert. Wird ein Destruktor aufgerufen, wird die Anzahl der existierenden Objekte heruntergesetzt.

```
<?php
  class Bruch{
    private $z=0;
    private $n=1;
    private static $anzahl=0;
```

Listing 4.15: Erweiterung um ein Klassenattribut und um eine Klassenmethode

```
    public function __construct($z,$n){
      $this->setZähler($z);
      $this->setNenner($n);
      Bruch::$anzahl++;
    }

    public function __destruct(){
      Bruch::$anzahl--;
    }

    public static function getAnzahl(){
      return Bruch::$anzahl;
  }

...
```

Listing 4.15: Erweiterung um ein Klassenattribut und um eine Klassenmethode (Forts.)

Ihnen fällt sicherlich der außergewöhnliche Zugriff auf das Klassenattribut über den *::*-Operator auf. Diesen Operator verwenden Sie immer dann, wenn Sie auf ein Klassenattribut oder auf eine Klassenmethode zugreifen. Der Zugriff erfolgt, indem Sie den Namen der Klasse selbst dem *::*-Operator voranstellen.

Die *$this*-Variable steht Ihnen in diesen Fällen nicht zur Verfügung, da sie ja eine Referenz auf ein existierendes Objekt darstellt. Die Anzahl kann jedoch auch ohne die Existenz eines Objekts zurückgegeben werden. In diesem Fall ist der Wert 0.

Listing 4.16 zeigt den Zugriff auf die Klassenmethode. Dies geschieht genauso wie im Konstruktor über die Klasse selbst. Nachdem der erste Bruch erzeugt und ausgegeben wurde, wird die Anzahl nochmals ausgegeben. Dies wird für einen zweiten Bruch wiederholt.

Da der Destruktor der Brüche erst „hinter" der letzten Zeile des PHP-Skripts ausgeführt wird, kann das Dekrementieren der Anzahl nicht dargestellt werden.

```
<?php require_once("classloader.inc.php"); ?>
<html><body>
<?php
  echo 'Anzahl: '.Bruch::getAnzahl().'<br>';
  $a=new Bruch(1,2);
  echo 'a: '.$a.'<br>';
  echo 'Anzahl: '.Bruch::getAnzahl().'<br>';
  $b=new Bruch(3,2);
  echo 'b: '.$b.'<br>';
```

Listing 4.16: Zugriff auf die statische Klassenmethode von außen

```
  echo 'Anzahl: '.Bruch::getAnzahl().'<br>';
?>
</body></html>
```

Listing 4.16: Zugriff auf die statische Klassenmethode von außen (Forts.)

Die Ausgabe erfolgt unter Verwendung der *__toString*-Methode aus dem vorherigen Kapitel und sieht folgendermaßen aus:

Anzahl: 0

a: 1/2

Anzahl: 1

b: 3/2

Anzahl: 2

Aus Sicht von gutem objektorientiertem Design ist von der Verwendung statischer Eigenschaften und Methoden abzuraten. Wenn Sie eine Vielzahl von statistischen Informationen verwalten wollen, sollten Sie eine eigene Verwaltungsklasse schreiben.

Diese Klasse ist dann intern für das Erstellen, Verwalten und Löschen von Brüchen verantwortlich. Sie beinhaltet ihrerseits nichtstatische Eigenschaften und Methoden und verwaltet eine Liste aller Brüche.

Bei der ausschließlichen Verwendung von Klassenattributen haben Sie beispielsweise ein Problem, wenn Sie ein zuvor serialisiertes Objekt wieder einlesen. Dabei wird kein Konstruktor aufgerufen und die Anzahl der Brüche im System nicht erhöht. In einer Verwaltungsklasse können Sie hingegen aus das Serialisieren und Deserialisieren verwalten und dabei auch die Anpassung der Anzahl von Objekten vornehmen.

4.1.6 Konstanten in Klassen und Verhinderung von Vererbung

Bereits in Kapitel 2.1 wurde gezeigt, wie Sie in PHP über den *define*-Befehl Konstanten in Ihr prozedurales Programm einbauen können. So definiert die folgende Zeile *define('WERT_KONST', 'Meine tolle Homepage');* eine Konstante mit dem Namen *WERT_KONST*, die den Inhalt *Meine tolle Homepage* enthält. Bei der Angabe des Namens einer Konstanten ist darauf zu achten, dass bei Konstanten zwischen Groß- und Kleinschreibung unterschieden wird.

Der *define*-Befehl funktioniert innerhalb einer Klasse jedoch nicht. Sie können also auf diese Weise keine Konstante als Eigenschaft einer Klasse definieren.

Stattdessen existiert für Klassen der Befehl *const*, mit dem eine für die Klasse gültige Konstante deklariert wird, die sich im Aufruf wie ein Klassenattribut verhält.

```
<?php
  class Kreis{
```

Listing 4.17: Eine Kreisklasse mit der Konstanten WertPI

```
    private $x; private $y; private $r;
    const wertPI=3.141592654;

    public function __construct($x,$y,$r){
      $this->x=$x; $this->y=$y; $this->r=$r;
    }

    public function getInhalt(){
      return Kreis::wertPI * $this->r * $this->r;
    }
  }
?>
```

Listing 4.17: Eine Kreisklasse mit der Konstanten WertPI (Forts.)

Listing 4.17 skizziert eine Klasse *Kreis*. Jeder Kreis besteht aus einem Mittelpunkt, der mit einer X/Y-Koordinate als Eigenschaft jedes Objekts beschrieben wird. Zusätzlich ist zur Beschreibung des Kreises die Angabe des Radius notwendig.

Wenn Sie Berechnungen mit Kreisen durchführen wollen, werden Sie häufig die Zahl PI verwenden müssen. Solche Zahlen werden gewöhnlich als Konstanten mit in der Klasse deklariert. In diesem Fall geschieht dies über die Anweisung *const wertPI=3.141592654;*.

Der Zugriff auf die Konstante erfolgt in der Methode *getInhalt()*, die den Flächeninhalt des Kreises berechnet. Hier sehen Sie, dass die Konstante genauso angesprochen wird wie ein Klassenattribut.

In Listing 4.18 wird die Methode *getInhalt()* aufgerufen, die den korrekten Flächeninhalt von etwas mehr als 78 Einheiten zurückgibt. Sie können auch auf die Konstante selbst so lesend zugreifen, als wäre sie eine statische Eigenschaft der Klasse.

```
<?php require_once("classloader.inc.php"); ?>
<html><body>
<?php
  $k=new Kreis(1,1,5);
  echo $k->getInhalt().'<br>';
  echo Kreis::wertPI.'<br>';
?>
</body></html>
```

Listing 4.18: Berechnung des Flächeninhalts und Zugriff auf die klasseninterne Konstante

Eine als Konstante definierte Eigenschaft können Sie einmalig setzen und danach nicht verändern.

Auch der Inhalt ganzer Methoden kann neu definiert werden, indem man eine neue Klasse von einer existierenden Klasse ableitet und eine Methode überschreibt

(Kap.3.2.2). Die Umsetzung des Überschreibens von Methoden in PHP wird in Kapitel 4.2.1 beschrieben.

Dieses Überschreiben können Sie jedoch verhindern, indem Sie eine Methode als *final* deklarieren. Eine gültige Deklaration wäre beispielsweise *final public function berechnen()*. Eine Unterkasse könnte diesen Methodeninhalt dann nicht neu deklarieren.

Wie in den meisten anderen objektorientierten Sprachen ist es in PHP 5 auch möglich, ganze Klassen vor Vererbung und damit vor der Redefinition von Methoden der vor Erweiterungen zu schützen. Mit einer finalen Klassendefinition, beispielsweise durch *final class Bruch* verhindern Sie, dass jemand von Ihrer Klasse eine Unterklasse ableitet. Sie verbieten also in diesem Fall die Vererbung gänzlich.

4.1.7 Referenzübergabe von Objekten und Kopien

In den neuen PHP-Versionen werden Objekte bei einem Funktionsaufruf und auch bei einer Zuweisung grundsätzlich als Referenz übergeben. Man übergibt also nicht eine Kopie des Objekts, wie das beispielsweise bei Zahlen üblich ist, sondern lediglich einen Zeiger auf das Objekt.

Dies ist auch bei anderen objektorientierten Sprachen üblich und auch sinnvoll, da Objekte im Speicher sehr viel Platz einnehmen können. Ein ständiges Kopieren von großen Objekten würde Performanceprobleme nach sich ziehen.

Wenn Sie jedoch unbedingt eine Kopie eines Objekts benötigen, bietet PHP Ihnen den *clone*-Befehl. Das Anlegen der Kopie wird von PHP automatisch durchgeführt. Sie können jedoch den Kopiervorgang eines Objekts einer Klasse beeinflussen, indem Sie die *__clone*-Methode in der Klassendefinition implementieren.

Listing 4.19 zeigt eine einfache Rechnungsklasse, die aus einer Rechnungssumme besteht und einem Merker, ob die Rechnung bereits ausgeliefert wurde oder nicht. Testweise wird eine weitere Eigenschaft hinzugefügt, die besagt, ob es sich bei der Rechnung um ein Original handelt oder nur um eine Kopie.

Über eine Set-Methode kann der Auslieferungsstatus der Rechnung verändert werden und die implementierte *__toString*-Methode gibt alle Daten der Rechnung als Zeichenkette aus. Abschließend ist noch die implementierte *__clone*-Methode zu nennen. Während beim regulären Klonvorgang alle Parameter des Originalobjekts auf die Kopie unverändert übertragen werden, werden in dieser Methode Änderungen des Vorgangs spezifiziert. In diesem Fall wird der Merker gesetzt, dass es sich bei dieser Rechnung um eine Kopie handelt.

```
<?php
  class Rechnung{
    private $summe; private $ausgeliefert=FALSE; private $istKopie=FALSE;
```

Listing 4.19: Eine einfache Rechnungsklasse

```
    public function __construct($summe){
      $this->summe=$summe;
    }

    public function setAusgeliefert($wert){
      if ($wert==FALSE){
        $this->ausgeliefert=FALSE;
      }
      else{
        $this->ausgeliefert=TRUE;
      }
    }

    public function __toString(){
      if ($this->ausgeliefert==TRUE)
        $ausgabe='Rechnung über '.$this->summe.'EUR wurde ausgeliefert.';
      else
        $ausgabe='Rechnung über '.$this->summe.'EUR wurde
                                          noch nicht ausgeliefert!';
      if ($this->istKopie==TRUE) $ausgabe.=' Ich bin ein Klon!';
      return $ausgabe;
    }

    public function __clone(){
      $this->istKopie=TRUE;
    }
  }
?>
```

Listing 4.19: Eine einfache Rechnungsklasse (Forts.)

Zusätzlich wird eine einfache Lagerklasse implementiert. Das Lager bekommt eine Rechnung, deren Inhalt sie ausliefern muss. Wenn dies geschehen ist, wird die Rechnung von dem Lager in den Status *ausgeliefert* gesetzt. In der Methode *ausliefern()* sehen Sie, wie der Datentyp des Eingabeparameters mithilfe von *instanceof* geprüft wird.

```
<?php
  class Lager{
    public function ausliefern($r){
      if ($r instanceof Rechnung){
        $r->setAusgeliefert(TRUE);
      }
```

Listing 4.20: Eine einfache Lagerklasse

```
    }
  }
?>
```

Listing 4.20: Eine einfache Lagerklasse (Forts.)

Im Fehlerfall findet in der *ausliefern*-Methode des Lagers noch keine Reaktion statt. Wie Sie in einer objektorientierten Sprache sinnvoll auf fehlerhafte Eingaben reagieren können, wird in Kapitel 4.3 beschrieben.

Listing 4.21 zeigt den Test der Rechnung und des Lagers, wobei die Referenzübergabe und das Klonen von Objekten am Beispiel der Rechnung durchgeführt werden.

```
<?php require_once("classloader.inc.php"); ?>
<html><body>
<?php
  $lager=new Lager();
  $r1=new Rechnung(100.0);
  $r2=$r1;
  $r3=clone $r1;
  $lager->ausliefern($r2);
  echo $r1.'<br>'; echo $r2.'<br>'; echo $r3.'<br>';
?>
</body></html>
```

Listing 4.21: Testen der Referenz und der Kopie

Zunächst wird eine Rechnung *$r1* angelegt. Die Variable *$r2* zeigt auf dieselbe Rechnung wie *$r1*, die beiden Referenzen/Zeiger haben also dasselbe Ziel. Nun wird eine weitere Referenz *$r3* angelegt. Deren Ziel ist ein Klon der Rechnung, auf die *$r1* (und auch *$r2*) zeigt. Der Klon ist jedoch ein eigenes, neues Objekt. Bei der Verwendung des Befehls *clone* wird dabei die Methode *__clone* in *$r2* ausgeführt, die die Eigenschaft *$istKopie* im Klon auf *TRUE* setzt. Abbildung 4.3 verdeutlicht den Sachverhalt zum Ende des PHP-Skripts nochmals.

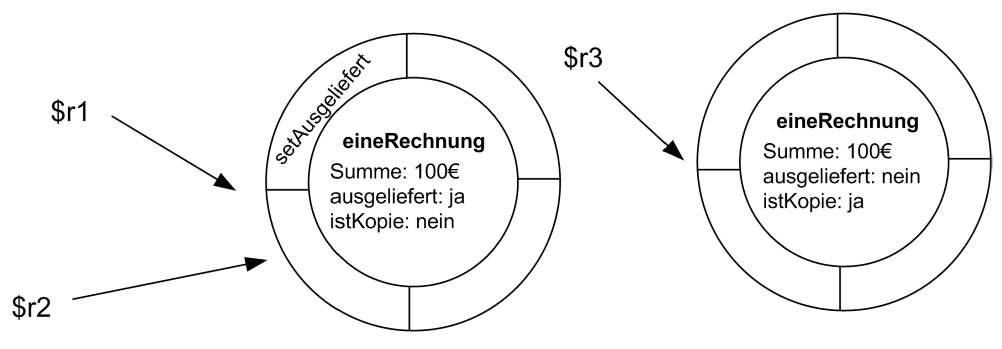


Abbildung 4.3: Objektreferenzen auf dasselbe Objekt und auf eine Kopie

Im Testprogramm wird zunächst die Kopie erstellt, bevor die Rechnung *$r2* (bzw. damit auch $r1) ausgeliefert wird. Die Kopie *$r3* ist deshalb immer noch im Zustand *nicht ausgeliefert*. Die Ausgabe des Testprogramms lautet also:

Rechnung über 100EUR wurde ausgeliefert.

Rechnung über 100EUR wurde ausgeliefert.

Rechnung über 100EUR wurde noch nicht ausgeliefert! Ich bin ein Klon!

4.1.8 Informationen über Objekte und Klassen zur Laufzeit

PHP bietet Ihnen die Möglichkeit an, zur Laufzeit weitere Informationen zu geladenen Klassen und zu existierenden Objekten abzufragen. Diese Informationen werden Metainformationen genannt.

Sie haben bereits den Befehl *instanceof* kennen gelernt, der in Verbindung mit einer Verzweigung prüft, ob ein Objekt eine Instanz einer bestimmten angegebenen Klasse ist oder nicht. Die folgende Tabelle listet die wichtigsten Metainformationen über Objekte auf.

Funktion	Bedeutung
$erg=get_class($obj)	gibt den Namen der Klasse des Objekts *$obj* zurück
$erg=get_parent_class($obj)	gibt den Namen der Ober-Klasse des Objekts *$obj* zurück oder *FALSE*, wenn keine Ober-Klasse existiert
$erg=method_exists($obj,$methode)	prüft, ob die Methode mit dem Namen *$methode* im Objekt *$obj* existiert oder nicht
$erg=is_subclass_of($obj,$klasse)	prüft, ob die Klasse des Objekts *$obj* eine Unterklasse von *$klasse* ist oder nicht
$erg=get_object_vars($obj)	gibt eine Liste der öffentlich sichtbaren Eigenschaften eines Objekts als Datenfeld zurück

Tabelle 4.1: Informationen über Objekte zur Laufzeit

In Listing 4.22 werden die Metainformationen auf der Rechnungsklasse, die in Listing 4.19 beschrieben wurde, getestet. Dabei wird zuerst eine Rechnung angelegt und dann werden die Informationen zu dieser Rechnung abgefragt.

```
<?php require_once("classloader.inc.php"); ?>
<html><body>
<?php
  $r1=new Rechnung(100.0);
  echo '<br>get_class:<br>';
  echo var_dump(get_class($r1));
  echo '<br>get_parent_class:<br>';
```

Listing 4.22: Testen der Metainformationen eines Objekts

```
  echo var_dump(get_parent_class($r1));
  echo '<br>method_exists:<br>';
  echo var_dump(method_exists($r1,'bla'));
  echo '<br>is_subclass_of:<br>';
  echo var_dump(is_subclass_of($r1,'Lager'));
  echo '<br>get_object_vars:<br>';
  echo var_dump(get_object_vars($r1));
?>
</body></html>
```

Listing 4.22: Testen der Metainformationen eines Objekts (Forts.)

Die Ausgabe der getesteten Methoden lautet wie erwartet:

get_class:

string(8) "Rechnung"

get_parent_class:

bool(false)

method_exists:

bool(false)

is_subclass_of:

bool(false)

get_object_vars:

array(0) { }

Zusätzlich dazu existieren noch einige Methoden, die Informationen zu einer Klasse selbst zurückgeben. Diese Methoden benötigen kein angelegtes Objekt. Diese Informationen können vor allem zur Fehlersuche hilfreich sein.

Funktion	Bedeutung
$erg=class_exists($klasse)	prüft, ob eine Klasse momentan geladen ist oder nicht
$erg=get_class_methods($klasse)	gibt die Namen der öffentlich zugänglichen Klassenmethoden in einem Datenfeld zurück
$erg=get_declared_classes()	gibt die Namen aller Klassen in einem Datenfeld zurück, die der PHP-Interpreter geladen hat
$erg=get_class_vars($klasse)	gibt die Namen der öffentlich zugänglichen Klassenattribute in einem Datenfeld zurück

Tabelle 4.2: Informationen über Klassen zur Laufzeit

Auch in diesem Fall werden die Methoden anhand der Rechnungsklasse getestet. Auf den dynamischen Klassenlader wird hier verzichtet, da kein Objekt angelegt wird. Die Klasse wird stattdessen manuell in der ersten Zeile des PHP-Skripts geladen.

```
<?php require_once("rechnung.inc.php"); ?>
<html><body>
<?php
  $klasse='Rechnung';
  echo '<br>class_exists:<br>';
  echo var_dump(class_exists($klasse));
  echo '<br>get_class_methods:<br>';
  echo var_dump(get_class_methods($klasse));
  echo '<br>get_declared_classes:<br>';
  echo var_dump(get_declared_classes());
  echo '<br>get_class_vars:<br>';
  echo var_dump(get_class_vars($klasse));
?>
</body></html>
```

Listing 4.23: Testen der Metainformationen einer Klasse

Die beeindruckende Ausgabe des Skripts aus Listing 4.23 lautet:

```
class_exists:
bool(true)
get_class_methods:
array(4) { [0]=> string(11) "__construct" [1]=> string(15) "setAusgeliefert" [2]=>
string(10) "__toString" [3]=> string(7) "__clone" }
get_declared_classes:
array(140) { [0]=> string(8) "stdClass" [1]=> string(9) "Exception" [2]=>
string(14) "ErrorException" [3]=> string(16) "COMPersistHelper" [4]=> string(13)
"com_exception" [5]=> string(19) "com_safearray_proxy" [6]=> string(7) "variant"
[7]=> string(3) "com" [8]=> string(6) "dotnet" [9]=> string(19)
"ReflectionException" [10]=> string(10) "Reflection" [11]=> string(26)
"ReflectionFunctionAbstract" [12]=> string(18) "ReflectionFunction" [13]=>
string(19) "ReflectionParameter" [14]=> string(16) "ReflectionMethod" [15]=>
string(15) "ReflectionClass" [16]=> string(16) "ReflectionObject" [17]=> string(18)
"ReflectionProperty" [18]=> string(19) "ReflectionExtension" [19]=> string(8)
"DateTime" [20]=> string(12) "DateTimeZone" [21]=> string(11) "LibXMLError" [22]=>
string(22) "__PHP_Incomplete_Class" [23]=> string(15) "php_user_filter" [24]=>
string(9) "Directory" [25]=> string(16) "SimpleXMLElement" [26]=> string(12)
"DOMException" [27]=> string(13) "DOMStringList" [28]=> string(11) "DOMNameList"
[29]=> string(21) "DOMImplementationList" [30]=> string(23)
"DOMImplementationSource" [31]=> string(17) "DOMImplementation" [32]=> string(7)
"DOMNode" [33]=> string(16) "DOMNameSpaceNode" [34]=> string(19)
"DOMDocumentFragment" [35]=> string(11) "DOMDocument" [36]=> string(11)
```

```
"DOMNodeList" [37]=> string(15) "DOMNamedNodeMap" [38]=> string(16)
"DOMCharacterData" [39]=> string(7) "DOMAttr" [40]=> string(10) "DOMElement" [41]=>
string(7) "DOMText" [42]=> string(10) "DOMComment" [43]=> string(11) "DOMTypeinfo"
[44]=> string(18) "DOMUserDataHandler" [45]=> string(11) "DOMDomError" [46]=>
string(15) "DOMErrorHandler" [47]=> string(10) "DOMLocator" [48]=> string(16)
"DOMConfiguration" [49]=> string(15) "DOMCdataSection" [50]=> string(15)
"DOMDocumentType" [51]=> string(11) "DOMNotation" [52]=> string(9) "DOMEntity"
[53]=> string(18) "DOMEntityReference" [54]=> string(24) "DOMProcessingInstruction"
[55]=> string(15) "DOMStringExtend" [56]=> string(8) "DOMXPath" [57]=> string(25)
"RecursiveIteratorIterator" [58]=> string(16) "IteratorIterator" [59]=> string(14)
"FilterIterator" [60]=> string(23) "RecursiveFilterIterator" [61]=> string(14)
"ParentIterator" [62]=> string(13) "LimitIterator" [63]=> string(15)
"CachingIterator" [64]=> string(24) "RecursiveCachingIterator" [65]=> string(16)
"NoRewindIterator" [66]=> string(14) "AppendIterator" [67]=> string(16)
"InfiniteIterator" [68]=> string(13) "RegexIterator" [69]=> string(22)
"RecursiveRegexIterator" [70]=> string(13) "EmptyIterator" [71]=> string(11)
"ArrayObject" [72]=> string(13) "ArrayIterator" [73]=> string(22)
"RecursiveArrayIterator" [74]=> string(11) "SplFileInfo" [75]=> string(17)
"DirectoryIterator" [76]=> string(26) "RecursiveDirectoryIterator" [77]=>
string(13) "SplFileObject" [78]=> string(17) "SplTempFileObject" [79]=> string(17)
"SimpleXMLIterator" [80]=> string(14) "LogicException" [81]=> string(24)
"BadFunctionCallException" [82]=> string(22) "BadMethodCallException" [83]=>
string(15) "DomainException" [84]=> string(24) "InvalidArgumentException" [85]=>
string(15) "LengthException" [86]=> string(19) "OutOfRangeException" [87]=>
string(16) "RuntimeException" [88]=> string(20) "OutOfBoundsException" [89]=>
string(17) "OverflowException" [90]=> string(14) "RangeException" [91]=> string(18)
"UnderflowException" [92]=> string(24) "UnexpectedValueException" [93]=> string(16)
"SplObjectStorage" [94]=> string(9) "XMLReader" [95]=> string(9) "XMLWriter" [96]=>
string(8) "SWFShape" [97]=> string(7) "SWFFill" [98]=> string(11) "SWFGradient"
[99]=> string(9) "SWFBitmap" [100]=> string(7) "SWFText" [101]=> string(12)
"SWFTextField" [102]=> string(7) "SWFFont" [103]=> string(14) "SWFDisplayItem"
[104]=> string(8) "SWFMovie" [105]=> string(9) "SWFButton" [106]=> string(9)
"SWFAction" [107]=> string(8) "SWFMorph" [108]=> string(9) "SWFSprite" [109]=>
string(8) "SWFSound" [110]=> string(11) "SWFFontChar" [111]=> string(16)
"SWFSoundInstance" [112]=> string(14) "SWFVideoStream" [113]=> string(15)
"SWFPrebuiltClip" [114]=> string(20) "mysqli_sql_exception" [115]=> string(13)
"mysqli_driver" [116]=> string(6) "mysqli" [117]=> string(14) "mysqli_warning"
[118]=> string(13) "mysqli_result" [119]=> string(11) "mysqli_stmt" [120]=>
string(15) "PDFlibException" [121]=> string(6) "PDFlib" [122]=> string(12)
"PDOException" [123]=> string(3) "PDO" [124]=> string(12) "PDOStatement" [125]=>
string(6) "PDORow" [126]=> string(10) "SoapClient" [127]=> string(7) "SoapVar"
[128]=> string(10) "SoapServer" [129]=> string(9) "SoapFault" [130]=> string(9)
"SoapParam" [131]=> string(10) "SoapHeader" [132]=> string(14) "SQLiteDatabase"
[133]=> string(12) "SQLiteResult" [134]=> string(16) "SQLiteUnbuffered" [135]=>
string(15) "SQLiteException" [136]=> string(13) "XSLTProcessor" [137]=> string(10)
"ZipArchive" [138]=> string(10) "paradox_db" [139]=> string(8) "Rechnung" }
get_class_vars:
array(0) { }
```

Beeindruckend ist, das PHP in der Version 5.2.9 bereits 139 Klassen geladen hat, ohne dass Sie selbst eine Klasse definiert haben. Auffallend sind insbesondere die Klassen für die XML-, PDF- und SQL-Unterstützung. Auf einige wenige dieser Klassen wird im weiteren Verlauf dieses Kapitels noch eingegangen. Im letzten, 139-sten Element des Datenfelds finden Sie dann Ihre geladene Rechnungsklasse.

4.2 Realisierung von Klassengeflechten

In diesem Kapitel stehen die Erstellung von mehreren Klassen und die Beziehung zwischen Objekten im Vordergrund. Im dritten Kapitel dieses Buches wurde gezeigt, wie Sie aus einer Geschäftsprozessanalyse, einer objektorientierten Analyse mit anschließendem Design Klassendiagramme erstellen, die in PHP 5 umgesetzt werden können.

Dabei kommen die folgenden objektorientierten Konzepte zum Einsatz:

- Vererbung und Polymorphie
- Assoziationen
- Aggregationen und Kompositionen
- Implementierung von Interfaces

Während in Kapitel 3 die Identifikation der Konzepte aus Problemstellungen Ihres Kunden im Vordergrund standen, fokussiert sich dieses Kapitel auf die Umsetzung der Konzepte in PHP 5.

4.2.1 Vererbung in PHP

In Kapitel 3.2.2 wurde bei der Analyse des Autohauses bereits die Vererbung angesprochen, die man an der „Ist ein“-Phrase erkennen kann. So ist der Verkäufer ein Mitarbeiter des Autohauses und jeder Mitarbeiter des Autohauses ist eine Person. Auch ein Kunde ist eine Person.

Es macht jedoch keinen Sinn, eine Person im System anzulegen, die weder ein Kunde noch ein Mitarbeiter ist. Die Klasse ist deshalb abstrakt definiert. Im Klassendiagramm der Abbildung 4.4 sind die drei Klassen mit einigen Eigenschaften und einer Methode abgebildet.

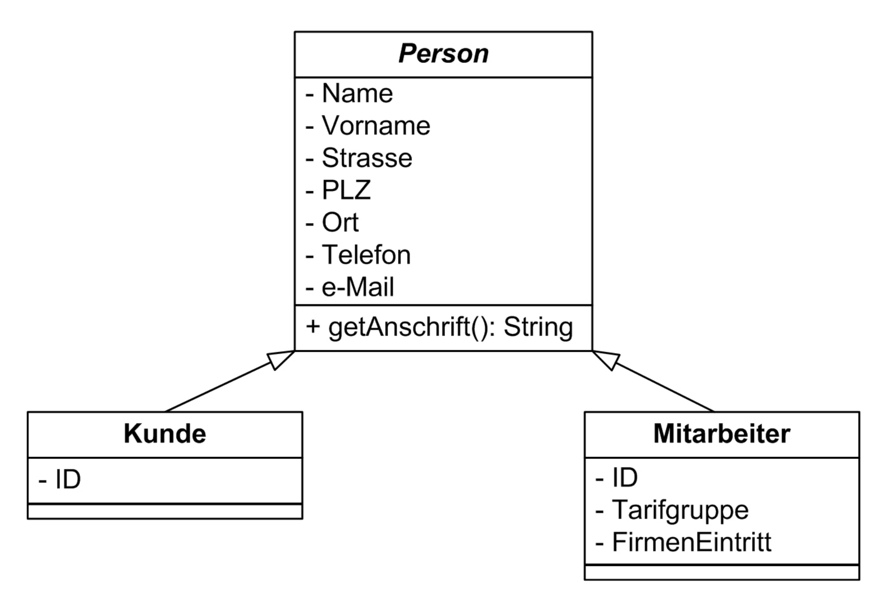


Abbildung 4.4: Klassendiagramm mit Vererbung und abstrakter Klasse

Es ist nun die Frage zu stellen, wie Sie das existierende Klassendiagramm aus der OOA und OOD in PHP 5 umsetzen können. Mit der Vererbung und abstrakten Klassen werden also weitere Elemente der objektorientierten Programmierung (OOP) vorgestellt.

```
<?php
  abstract class Person{
    private $name; private $vorname;
    private $strasse; private $plz; private $ort;
    private $tel; private $mail;

    public function __construct($name,$vorname,$strasse,$plz,$ort){
      $this->name=$name; $this->vorname=$vorname;
      $this->strasse=$strasse; $this->plz=$plz; $this->ort=$ort;
    }

    public function getName(){
      return $this->name;
    }
    public function setName($name){
      $this->name=$name;
    }
    ....
    public function getAnschrift(){
```

Listing 4.24: Die abstrakte Klasse „Person"

```
      $ret='';
      $ret.=$this->name.' '.$this->vorname.'<br>';
      $ret.=$this->strasse.'<br>';
      $ret.=$this->plz.' '.$this->ort.'<br>';
      return $ret;
    }
  }
?>
```

Listing 4.24: Die abstrakte Klasse „Person“ (Forts.)

Eine abstrakte Klasse, von der man keine Objekte anlegen kann, wird einfach durch das Schlüsselwort *abstract* vor der Klassendefinition erzeugt. Im Konstruktor werden die üblichen Muss-Felder für die Erzeugung einer Person aus einer abgeleiteten Klasse definiert. Zusätzlich werden die üblichen Get- und Set-Methoden definiert, ebenso die geforderte Methode *getAnschrift()*.

Profitipp

Bei einer komplexeren Rückgabe ist es sinnvoll, den Wert der Rückgabe über eine Hilfsvariable (hier: *$ret*) zusammenzusetzen. So spart man unnötig lange Codezeilen.

Wenn Sie versuchen, in einer Testklasse über *$a=new Person(...);* ein Personen-Objekt anzulegen, werden Sie folgende Fehlermeldung erhalten:

Fatal error: Cannot instantiate abstract class Person in ...

Hinweis

Während man durch *final* die weitere Vererbung verbietet, zwingt man durch *abstract* den Programmierer zu einer Vererbungshierarchie.

Nun sollten Sie eine Klasse ableiten, von der man konkrete Objekte erzeugen kann. Das Schlüsselwort dazu lautet *extends*, gefolgt von der Klasse, von der man ableiten will. Im Konstruktor der Unterklasse kann man den Konstruktor der Oberklasse bzw. der Elternklasse über *parent::__construct(...)* ähnlich wie eine Klassenmethode aufrufen. Dort werden dann erst einmal die Daten gespeichert, die zu jeder Person gehören. Der neu erzeugte Kunde merkt sich im Anschluss daran noch seine Kunden-ID, die über die passende Get-Methode ausgelesen werden kann.

```
<?php
  class Kunde extends Person{
    private $id;
```

Listing 4.25: Die abgeleitete konkrete Klasse „Kunde“

```
    public function __construct($id,$name,$vorname,$strasse,$plz,$ort){
      parent::__construct($name,$vorname,$strasse,$plz,$ort);
      $this->id=$id;
    }

    public function getID(){
      return $this->id;
    }
  }
?>
```

Listing 4.25: Die abgeleitete konkrete Klasse „Kunde" (Forts.)

Die zweite Klasse, die von der Person abgeleitet werden soll, ist die Klasse der Mitarbeiter. Dabei wird in gleicher Weise wie bei der Kundenklasse vorgegangen. Nur die für einen Mitarbeiter typischen Eigenschaften werden auch in der Mitarbeiterklasse festgehalten.

Sie erkennen auch, dass alle Eigenschaften, die Mitarbeiter und Kunden gemeinsam haben, ausschließlich in der Personenklasse gespeichert und verwaltet werden. Auf diese Weise wird doppelter Quellcode verhindert und die Wartbarkeit der Anwendung verbessert.

```
<?php
  class Mitarbeiter extends Person{
    private $id; private $tarifGruppe;
    private $firmenEintritt;

    public function __construct($id,$name,$vorname,$strasse,$plz,$ort,
                                       $tarifGruppe,$firmenEintritt){
      parent::__construct($name,$vorname,$strasse,$plz,$ort);
      $this->id=$id; $this->tarifGruppe=$tarifGruppe;
      $this->firmenEintritt=$firmenEintritt;
    }

    public function getID(){
      return $this->id;
    }
  }
?>
```

Listing 4.26: Die abgeleitete konkrete Klasse „Mitarbeiter"

Sowohl der Mitarbeiter als auch der Kunde verfügen über die Eigenschaft *$id*. Wenn alle konkreten Personen über eine ID verfügen, wieso wird diese ID dann nicht in die Ober-

klasse ausgelagert? In PHP ist dies durchaus eine Designalternative, da der Datentyp einer Eigenschaft nicht im Vorfeld festgelegt werden muss. Es können jedoch auch hier Probleme auftreten, wenn man in einer Set-Methode die Gültigkeit einer ID prüfen muss. Denn der Identifikator kann bei einem Kunden völlig anders aufgebaut sein als bei einem Mitarbeiter. Ein Lösungsansatz würde für PHP darin bestehen, die Eigenschaft *$id* zentral für den Kunden *protected* anstatt *private* zu definieren. Damit könnten auch Methoden der Unterklasse direkt auf die Eigenschaft zugreifen. Im zweiten Schritt könnten dann die Set-Methoden des Kunden und des Mitarbeiters mit den entsprechenden Prüfungen überschrieben werden. Wie man Methoden überschreibt, wird im nächsten Beispiel erläutert.

In Listing 4.27 werden zunächst die drei erzeugten Klassen getestet. Es werden ein Kunde und ein Mitarbeiter erzeugt. Weil beide auch Personen sind, besitzen sie die Methode *getAnschrift()*, die in beiden Fällen aufgerufen wird. Abschließend wird testweise der PHP-Befehl *get_parent_class* auf der Referenz des Mitarbeiterobjekts angewendet. Sie gibt den Namen der Oberklasse des Mitarbeiters, also *Person* zurück.

```
<?php require_once("classloader.inc.php"); ?>
<html><body>
<?php
  $a=new Kunde(1,'Frank','Dopatka','Hauptstrasse 4',51580,'Reichshof');
  echo $a->getAnschrift();
  $b=new Mitarbeiter('E34','Ulli','Streber','Sackgasse 2',12234,'Berlin',
                                                  'TvöD13','31.01.2008');
  echo $b->getAnschrift();
  echo get_parent_class($b);
?>
</body></html>
```

Listing 4.27: Testklasse der Vererbung

Die Ausgabe der Testklasse lautet wie erwartet:

Frank Dopatka

Hauptstrasse 4

51580 Reichshof

Ulli Streber

Sackgasse 2

12234 Berlin

Person

Im zweiten Beispiel der Vererbung wird die Tierhierarchie aus dem dritten Kapitel umgesetzt. Dort wurden im objektorientierten Design die Klassen aus Abbildung 4.5 ermittelt.

Die abstrakte Klasse *Tier* hat eine Eigenschaft *name*, die als *protected* deklariert ist. Somit kann der Zugriff für abgeleitete Klassen wie auf eine *public*-deklarierte Eigenschaft erfolgen.

Zusätzlich dazu ist auch die Methode *gibLaut()* im Tier laut der Definition im Klassendiagramm der UML abstrakt definiert. Man möchte also, dass jedes konkrete Tier Laut geben kann. Wie dieser Laut jedoch aussieht, hängt von dem konkreten Tier ab.

Abgeleitet von der Tierklasse werden die Klassen *Hund* und *Katze*. Sie implementieren das Lautgeben und zusätzlich noch die eigenen Methoden *bellen()* und *miauen()*. Jemand, der einen Hund verwaltet, möchte den konkreten Laut unter Umständen direkt abfragen.

Zusätzlich ist eine Dogge ein spezieller Hund. Er kann besonders gut beißen und bietet dies als eigene Methode an.

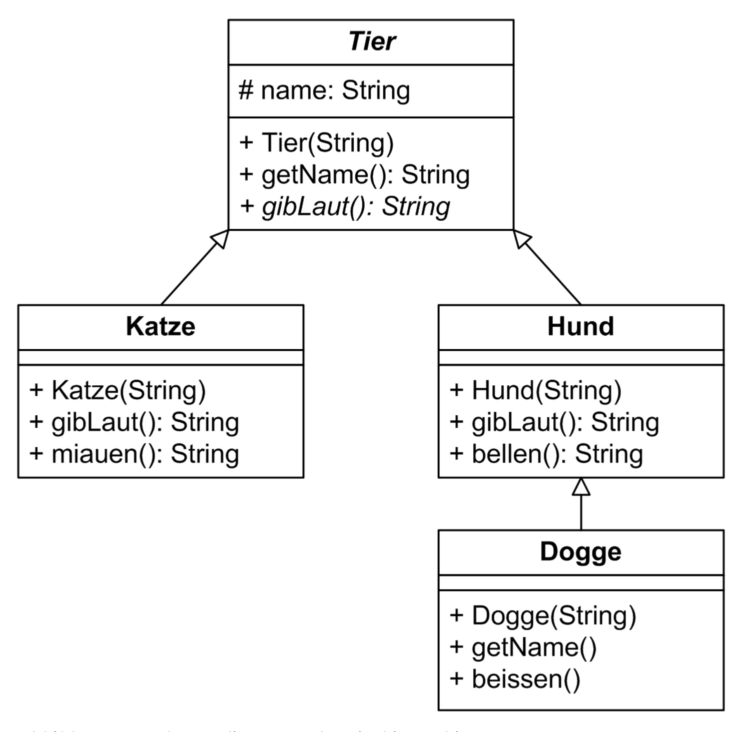


Abbildung 4.5: Klassendiagramm der Tierhierarchie

Listing 4.28 zeigt die Umsetzung der Tierklasse in PHP 5. Beachten Sie die Syntax der abstrakten Methode *gibLaut()*.

```
<?php
  abstract class Tier{
    protected $name;
```

Listing 4.28: Die abstrakte Klasse „Tier“ mit abstrakter Methode „gibLaut“

```
    public function __construct($name){
      $this->name=$name;
    }

    public function getName(){
      return $this->name;
    }

    abstract public function gibLaut();
  }
?>
```

Listing 4.28: Die abstrakte Klasse „Tier" mit abstrakter Methode „gibLaut" (Forts.)

Im nächsten Schritt wird die Katze modelliert, die ein Tier ist. Im Konstruktor besteht die Möglichkeit, die Eigenschaft *name* direkt zu setzen. Darauf wird jedoch verzichtet und stattdessen der Konstruktor der Oberklasse aufgerufen. Dort könnten zukünftig noch weitere Initialisierungen vorgenommen werden, die auch für ein Katzenobjekt von Bedeutung wären.

Neben der Bereitstellung der Methode *miauen()* ist die Implementierung der Methode *gibLaut()* zu nennen. Die Methodendefinition muss genauso lauten wie in der Oberklasse bis auf das fehlende Schlüsselwort *abstract*. Dadurch, dass diese Methode nun realisiert wurde, können konkrete Objekte der Klasse angelegt werden.

```
<?php
  class Katze extends Tier{

    public function __construct($name){
      parent::__construct($name);
    }

    public function gibLaut(){
      return $this->miauen();
    }

    public function miauen(){
      return 'miau';
    }
  }
?>
```

Listing 4.29: Die abgeleitete Klasse „Katze"

Die Implementierung der Hundeklasse erfolgt in Listing 4.30 analog zur Katzenklasse.

```
<?php
  class Hund extends Tier{

    public function __construct($name){
      parent::__construct($name);
    }

    public function gibLaut(){
      return $this->bellen();
    }

    public function bellen(){
      return 'wow';
    }
  }
?>
```

Listing 4.30: Die abgeleitete Klasse „Hund“

Von der Hundeklasse wird wiederum die Dogge als spezieller Hund abgeleitet. Da der Hund bereits das Lautgeben implementiert, ist dies bei der Dogge nicht zwingend nötig. Der Laut könnte natürlich durch ein Überschreiben der Methode mit identischer Definition *public function gibLaut()* neu definiert werden. Stattdessen wird die Methode *getName()* bei der Dogge im Vergleich zum Tier neu definiert. Zusätzlich wird die neue Methode *beissen()* definiert.

```
<?php
  class Dogge extends Hund{

    public function __construct($name){
      parent::__construct($name);
    }

    public function beissen(){
      return 'grrr...';
    }

    public function getName(){
      return $this->name.' die Dogge.';
    }
  }
?>
```

Listing 4.31: Die Dogge als spezieller Hund

Der Test der Tierklassen erfolgt in Listing 4.32. Dabei wird ein Datenfeld angelegt, das mit einer Katze und einer Dogge gefüllt wird. Beide geben ihren Namen aus, dann beißt die Dogge einmal zu. Da die Dogge die Methode *getName()* neu definiert hat, wird die neue Methode ausgeführt.

Anschließend soll jedes Tier im Datenfeld einen Laut von sich geben. Dazu wird das Feld in einer Schleife durchlaufen. Dies ist natürlich besonders bei großen Feldern von Interesse. Mit *instanceof* sollte noch eine zusätzliche Typprüfung durchgeführt werden, ob es sich bei den Elementen des Feldes wirklich um Tiere handelt.

Zum Abschluss wird versucht, jedes Tier im Datenfeld zum Miauen zu bringen. Bei der Dogge sollte dies problematisch werden.

```
<?php require_once("classloader.inc.php"); ?>
<html><body>
<?php
  $zoo=Array();
  $zoo[0]=new Katze('Nicki');
  echo $zoo[0]->getName().'<br>';
  $zoo[1]=new Dogge('Hasso');
  echo $zoo[1]->getName().'<br>';
  echo $zoo[1]->beissen().'<br>';
  for ($i=0;$i<2;$i++){
    echo $zoo[$i]->gibLaut().'<br>';
  }
  for ($i=0;$i<2;$i++){
    echo $zoo[$i]->miauen().'<br>';
  }
?>
</body></html>
```

Listing 4.32: Testklasse für verschiedene Tiere

Die Ausgabe ist auch hier erwartungsgemäß. Interessant ist auch die Fehlermeldung, dass eine Dogge nicht miauen kann.

Nicki

Hasso die Dogge.

grrr...

miau

wow

miau

Fatal error: Call to undefined method Dogge::miauen() in ...

4.2.2 Aufbau von Bekanntschaften: Assoziationen

Die Vererbung ist eine statische Abhängigkeit von Klassen, die zur Entwicklungszeit festgelegt wird. Die Bekanntschaft von Objekten untereinander entsteht jedoch erst zur Laufzeit. Nachdem ein Objekt ein anderes Objekt kennt, kann es Methoden auf dem bekannten Objekt ausführen.

Als Beispiel wird die Beziehung einer Rechnungsposition zu einem Artikel verwendet. Das Klassendiagramm der Abbildung 4.6 gibt dabei die Eigenschaften beider Klassen vor. Aus Sicht des Artikels ist die Bindung sehr lose; ein Artikel kennt keine Rechnungspositionen. Er kann aber auf vielen Rechnungspositionen enthalten sein.

Eine Rechnungsposition kennt stets genau einen Artikel, der in einer gewissen Menge eingekauft werden soll. Diese Abhängigkeit besteht bereits beim Erstellen der Rechnungsposition. Zusätzlich wird der Einzelpreis des Artikels und ggf. ein Rabatt auf die jeweilige Rechnungsposition festgehalten.

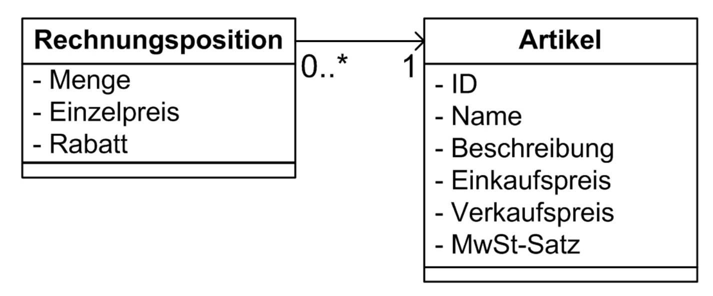


Abbildung 4.6: Beziehung zwischen einer Rechnungsposition und einem Artikel

Listing 4.33 zeigt die Implementierung einer Rechnungsposition. Der Artikel ist dabei eine Eigenschaft der Rechnungsposition und wird als Referenz im Konstruktor übergeben. Der Einzelpreis der Rechnungsposition wird aus dem Verkaufspreis des Artikels ermittelt. Außerdem werden dir typischen Get-Methoden implementiert. Der Gesamtpreis ist keine eigene Eigenschaft, sondern wird aus den existierenden Eigenschaften unter Berücksichtigung des optionalen Rabatts auf die Rechnungsposition sowie der Mehrwertsteuer aus dem Artikel berechnet.

```
<?php
  class Rechnungsposition{
    private $artikel; private $menge; private $ep; private $rabatt; // %

    public function __construct($artikel,$menge,$rabatt){
      $this->artikel=$artikel; $this->menge=$menge;
      $this->ep=$artikel->getVK(); $this->rabatt=$rabatt;
    }

    public function getArtikel(){
```

Listing 4.33: Die Klasse „Rechnungsposition"

```
      return $this->artikel;
    }
...
    public function getGesamtpreis(){
      return (($this->ep)*($this->menge))*(1-($this->rabatt)/100)*
                                (1+$this->artikel->getMwST()/100);
    }
  }
?>
```

Listing 4.33: Die Klasse „Rechnungsposition" (Forts.)

Die in Listing 4.34 implementierte Artikelklasse hat keine Besonderheiten, sie implementiert lediglich die aus dem Klassendiagramm geforderten Eigenschaften und bietet Get-Methoden dazu an.

```
<?php
  class Artikel{
    private $id; private $name; private $beschreibung;
    private $ek; private $vk; private $mwst;

    public function __construct($id,$name,$beschreibung,$ek,$vk,$mwst){
      $this->id=$id; $this->name=$name;
      $this->beschreibung=$beschreibung;
      $this->ek=$ek; $this->vk=$vk; $this->mwst=$mwst;
    }

    public function getId(){
      return $this->id;
    }
...
    public function getMwSt(){
      return $this->mwst;
    }
  }
?>
```

Listing 4.34: Die Klasse „Artikel"

Abbildung 4.7 zeigt nun ein Objektdiagramm des kleinen Testprogramms, das in Listing 4.33 vorgestellt wird. Da das Objektdiagramm ein konkretes Beispiel aus der objektorientierten Analyse bildet, aus dem die Klassendiagramme erst abgeleitet werden (Kap. 3), muss dieses Beispiel jetzt nachgebildet werden können.

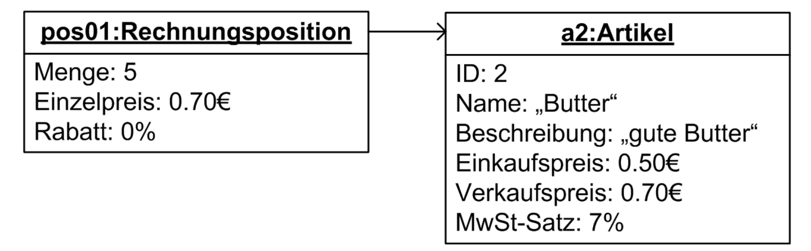


Abbildung 4.7: Objektdiagramm einer Rechnungsposition mit einem Artikel

Zunächst wird der Artikel erstellt und danach die Rechnungsposition. Diese erhält beim Konstruktoraufruf die Referenz auf den Artikel. Interessant ist auch die Möglichkeit, über die Rechnungsposition den Artikel zu ermitteln und dann eine Methode dieses Artikels, in diesem Fall *getBeschreibung()*, aufzurufen. Dieser Aufruf kann über *$pos01->getArtikel()->getBeschreibung()* in einer Zeile erfolgen.

```
<?php require_once("classloader.inc.php"); ?>
<html><body>
<?php
  $a2=new Artikel(2,'Butter','gute Butter',0.50,0.70,7);
  $pos01=new Rechnungsposition($a2,5,0);
  echo 'Name: '.$pos01->getArtikel()->getName().'<br>';
  echo 'Beschreibung: '.$pos01->getArtikel()->getBeschreibung().'<br>';
  echo 'Menge: '.$pos01->getMenge().'<br>';
  echo 'Einzelpreis: '.number_format($pos01->getEP(),2).' EUR<br>';
  echo 'Gesamtpreis: '.number_format($pos01->getGesamtpreis(),2).
                                                              ' EUR<br>';
?>
</body></html>
```

Listing 4.35: Test einer Rechnungsposition und eines Artikels

Somit lautet die Ausgabe des Testprogramms wie folgt:

Name: Butter

Beschreibung: gute Butter

Menge: 5

Einzelpreis: 0.70 EUR

Gesamtpreis: 3.75 EUR

4.2.3 Wechselseitige Bekanntschaften

Schwieriger zu realisieren ist die wechselseitige Assoziation zweier Klassen. Als Beispiel wird die Beziehung zwischen Studenten und Praktika realisiert. Da ein Student eine spezielle Person ist, kann die implementierte Personenklasse aus Listing 4.24 weiterverwendet werden.

Profitipp

Sie erkennen hier, dass eine sinnvoll durchdachte abstrakte Klasse auch in verschiedenem Kontext – in Listing 4.24 war der Kontext die Ableitung von Kunden und Mitarbeitern – weiterverwendet werden und damit langfristig Ressourcen der Implementierung einsparen kann.

Im Gegensatz zu einer Person besitzt ein Student zusätzlich eine Matrikelnummer und ein Datum der Immatrikulation. Ein Student kann sich an bis zu 5 Praktika gleichzeitig anmelden. Man kann einen Studenten fragen, an welchen Praktika er teilnimmt. Er gibt daraufhin ein Datenfeld seiner Praktikumsobjekte zurück.

Nach einer Anmeldung soll der Student sein Praktikum kennen, er muss ja wissen, an welchen Veranstaltungen er teilnimmt, und gleichzeitig muss auch das Praktikum seine Studenten kennen, da ggf. eine Anwesenheitspflicht besteht.

Ein Praktikum hat einen Namen und eine Zeitdauer, in dem es stattfindet. Zu einem Praktikum können sich bis zu 20 Studenten anmelden. Eine andere Sichtweise besteht darin, dass sich ein Student an einem Praktikum anmeldet. Beide Klassen verfügen also über eine Methode *anmelden*.

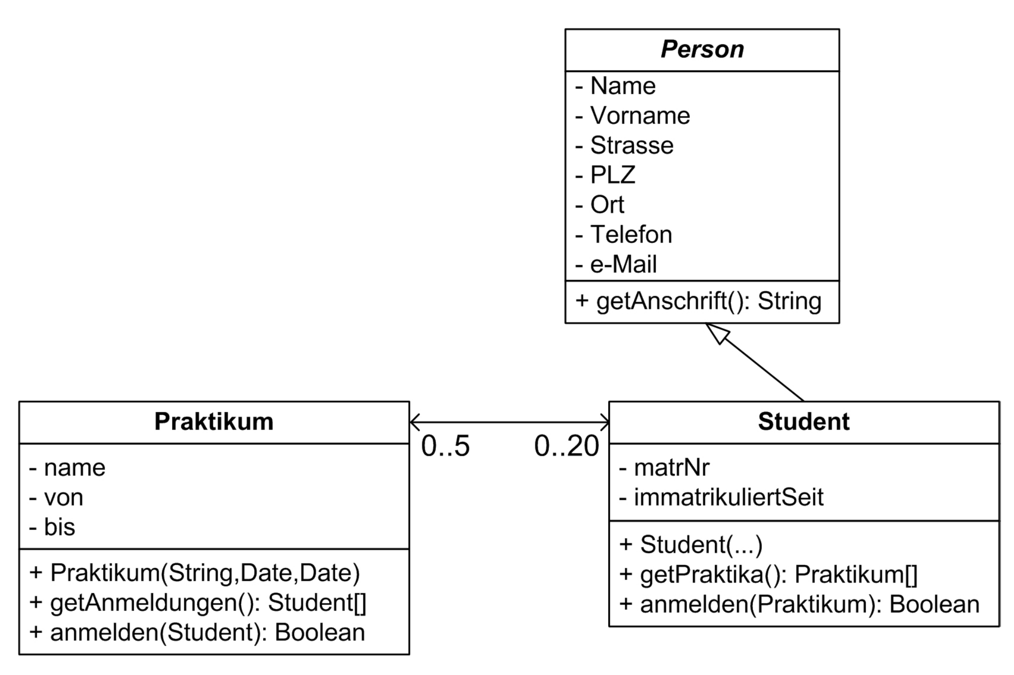


Abbildung 4.8: Wechselseitige Beziehung zwischen Praktika und Studenten

Hinweis

Die Anzahl der Eigenschaften und der Methoden ist in diesen Beispielen natürlich beschränkt, sodass Sie die Testprogramme noch nicht in der Realität einsetzen können. Es geht vielmehr darum, dass Sie den Quellcode verstehen, der die Beziehungen zwischen den Klassen realisiert und dass Sie diesen Code auch bei der Lösung anderer Problemstellungen einsetzen können.

Der Vorgang der Anmeldung ist also der Kern dieses Problems. Ein Student kann sich bei einem Praktikum anmelden und umgekehrt. Im Anschluss an eine Anmeldung müssen die Objekte beider Klassen eine Referenz auf das jeweils andere Objekt besitzen. Hier besteht die Gefahr einer endlos laufenden Rekursion.

Das Aktivitätsdiagramm auf Muschelebene der Abbildung 4.9 zeigt die Anmeldung eines Studenten *s* an einem Praktikum *p* mit der Methode *s.anmelden(p)*. Der umgekehrte Fall *p.anmelden(s)* verläuft analog. Die Pfeile des regulären Ablaufs sind etwas dicker dargestellt. Dies ist nicht in der UML spezifiziert, erhöht jedoch die Übersichtlichkeit.

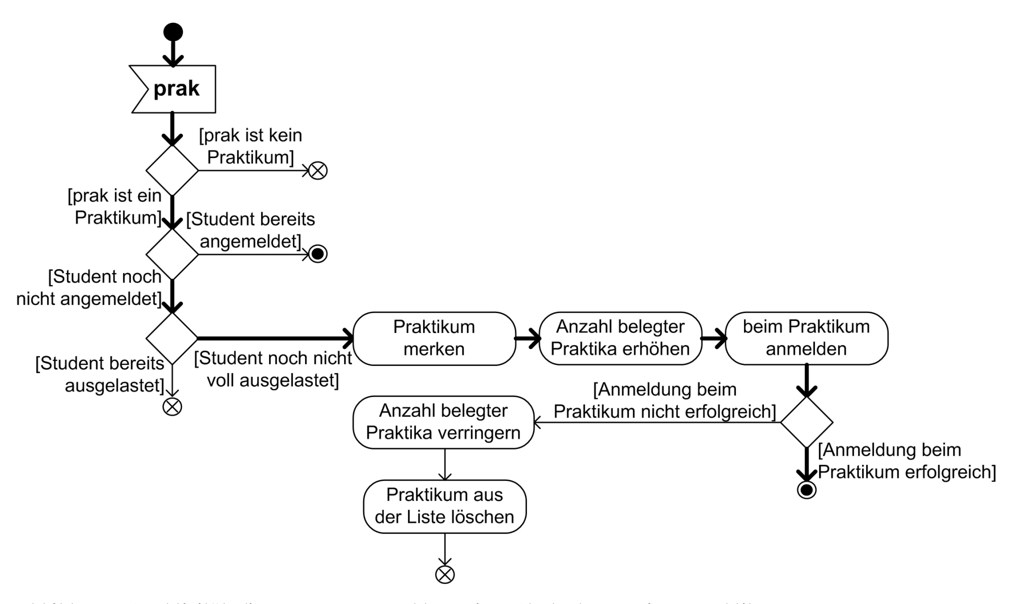


Abbildung 4.9: Aktivitätsdiagramm zur Anmeldung eines Studenten an einem Praktikum

Zunächst wird ein Praktikumsobjekt an die *anmelden*-Methode des Studenten übergeben. Da PHP die Datentypen nicht im Vorfeld festlegt, muss man zuerst prüfen, ob es sich bei der Eingabe um ein Praktikum handelt.

Im zweiten Schritt wird geprüft, ob der Student bereits an diesem Praktikum angemeldet ist. Wenn dies der Fall ist, ist der Anmeldevorgang bereits erfolgreich beendet.

Da sich ein Student gleichzeitig für maximal 5 Praktika anmelden kann, wird bei der dritten Prüfung ermittelt, ob die Obergrenze schon im Vorfeld erreicht ist. Ist dies nicht

der Fall, hat der Student noch freie Kapazitäten und kann sich die Referenz auf das Praktikum merken. Dann wird die Anzahl der belegten Praktika erhöht.

Bis zu diesem Zeitpunkt weiß das Praktikum unter Umständen noch nichts von der Anmeldung. Daher versucht der Student nun, sich beim Praktikum anzumelden. Wenn dies erfolgreich ist, wurde der gesamte Anmeldevorgang erfolgreich abgeschlossen. Andernfalls ist die Anzahl der belegten Praktika wieder zu verringern, das Praktikum wieder aus der Liste der besuchten Praktika zu entfernen und der Anmeldevorgang ist fehlgeschlagen.

Es stellt sich die Frage, warum der Student sich zuerst die Praktikumsreferenz merkt und dann bei einem Fehlschlag bei der Anmeldung am Praktikum die Referenz wieder löscht. Dies ist notwendig, da bei der Anmeldung am Praktikum intern wieder die eigene *anmelden*-Methode aufgerufen wird und die Gefahr der endlosen Rekursion besteht. Bei dieser Lösung gibt die Anmeldung des Studenten beim zweiten Aufruf jedoch auf Grund der zweiten Prüfung im Aktivitätsdiagramm *TRUE* zurück, noch bevor die Rekursion *beim Praktikum anmelden* noch einmal aufgerufen wird.

Diese textuelle Beschreibung, die wiederum aus dem Aktivitätsdiagramm resultiert, wird nun in Listing 4.36 in Quellcode gegossen.

```
<?php
  class Student extends Person{
    private $matrNr; private $immatrikuliertSeit;
    private $praks=Array(); private $anzPraks=0; const MAX_PRAKS=5;

    public function __construct($matrNr,$name,$vorname,$strasse,$plz,$ort,
                                              $immatrikuliertSeit){
    parent::__construct($name,$vorname,$strasse,$plz,$ort);
    $this->matrNr=$matrNr;
    $this->immatrikuliertSeit=$immatrikuliertSeit;
  }

    public function GetMatrNr(){
      return $this->matrNr;
    }
    public function getImmatrikuliertSeit(){
      return $this->immatrikuliertSeit;
    }
    public function getPraktika(){
      return $this->praks;
    }

    public function anmelden($prak){
```

Listing 4.36: Die Klasse „Student“

```
      if (($prak instanceof Praktikum)==FALSE) return FALSE;
      foreach (($this->praks) as $elem => $wert){
        if ($wert==$prak) return TRUE; // bereits angemeldet
      }
      if ($this->anzPraks==Student::MAX_PRAKS) return FALSE;
      // anmelden...
      $this->praks[$this->anzPraks]=$prak;
      $this->anzPraks++;
      // beim Praktikum anmelden
      if (($prak->anmelden($this))==FALSE){
        $this->anzPraks--;
        $this->praks[$this->anzPraks]=NULL;
        return FALSE;
      }
      return TRUE;
    }
  }
?>
```

Listing 4.36: Die Klasse „Student" (Forts.)

Die Eigenschaften Matrikelnummer und das Datum der Immatrikulation sind nicht die einzigen Eigenschaften, die ein Student mehr besitzt als eine gewöhnliche Person. Zusätzlich muss auch die Beziehung zu den belegten Praktika festgehalten werden. Dazu wird ein Datenfeld *$praks* definiert, as die Referenzen auf die Praktikumsobjekte enthalten soll. Zusätzlich wird die Anzahl der Praktika gespeichert, an denen der Student aktuell teilnimmt sowie die maximal mögliche Anzahl der Praktika als Konstante. Der Konstruktor und die Get-Methoden enthalten keinen spannenden Quellcode.

Die *anmelden*-Methode prüft mit *instanceof* zunächst den Datentyp des übergebenen Eingabeparameters. Ist es kein Praktikum, so ist die Anmeldung fehlgeschlagen. Dann wird die Liste der Praktika in einer *foreach*-Schleife durchgegangen, die der Student bereits besucht. Ist die übergebene Referenz dort bereits vorhanden, war der Student bereits angemeldet und der Anmeldevorgang endet erfolgreich. Dann wird geprüft, ob der Student bereits seine maximale Anzahl an Praktika gespeichert hat. Wenn dies der Fall ist, darf er sich nicht mehr für ein weiteres Praktikum anmelden.

Dann beginnt die eigentliche Anmeldung, bei der der Zähler der Teilnahmen erhöht und die Referenz auf das Praktikum gespeichert wird. Dann muss das Praktikum noch über die Teilnahme dieses Studenten informiert werden. Der Student ruft also seinerseits den Anmeldevorgang des Praktikums auf. Dabei muss ein Studentenobjekt übergeben werden, also das eigene Objekt. Dies ist möglich durch die Übergabe der *$this*-Referenz, die ja eine Referenz auf das Objekt selbst darstellt. Wenn diese Anmeldung fehlschlägt, wird die eigene Anmeldung rückgängig gemacht und der Vorgang ist fehlgeschlagen. Andernfalls ist die Anmeldung erfolgreich verlaufen.

Hinweis

Vergleichen Sie das Aktivitätsdiagramm aus Abbildung 4.9 mit der *anmelden*-Methode aus Listing 4.36. Erkennen Sie die Übereinstimmung?

Im Anschluss daran wird in Listing 4.37 die Praktikumsklasse vorgestellt mit ihren Eigenschaften des Namens sowie des Zeitraums, in dem das Praktikum stattfindet. Die Implementierung ist identisch zur Studentenklasse.

```
<?php
  class Praktikum{
    private $name; private $von; private $bis;
    private $studs=Array(); private $anzStuds=0; const MAX_STUDS=20;

    public function __construct($name,$von,$bis){
      $this->name=$name; $this->von=$von; $this->bis=$bis;
    }

    public function GetName(){
      return $this->name;
    }
    public function getVon(){
      return $this->von;
    }
    public function getBis(){
      return $this->bis;
    }
    public function getStudenten(){
      return $this->studs;
    }

    public function anmelden($stud){
      if (($stud instanceof Student)==FALSE) return FALSE;
      foreach (($this->studs) as $elem => $wert){
        if ($wert==$stud) return TRUE; // bereits angemeldet
      }
      if ($this->anzStuds==Praktikum::MAX_STUDS) return FALSE;
      // anmelden...
      $this->studs[$this->anzStuds]=$stud;
      $this->anzStuds++;
      // dem Studenten Bescheid sagen...
```

Listing 4.37: Die Klasse „Praktikum“

```
      if (($stud->anmelden($this))==FALSE){
        $this->anzStuds--;
        $this->studs[$this->anzStuds]=NULL;
        return FALSE;
      }
      return TRUE;
    }
  }
?>
```

Listing 4.37: Die Klasse „Praktikum" (Forts.)

Bei einer Vererbung, wie sie beispielsweise zwischen *Person* und *Student* existiert, ist die Abhängigkeit des Studenten von der Personenklasse größer als bei der Beziehung der Studenten und Praktika, da die Unterklasse nicht ohne ihre Oberklasse ausgeführt werden kann. Die Personenklasse benötigt die Studentenklasse jedoch nicht.

Der Quellcode des Listings 4.38 testet die beiden Klassen. Dabei werden zwei Studenten und drei Praktika angelegt. Es wird sowohl die Anmeldemethode der Studentenklasse als auch die Anmeldemethode des Praktikums getestet. So wird die korrekte Funktion in beiden Fällen sichergestellt.

```
<?php require_once("classloader.inc.php"); ?>
<html><body>
<?php
  $s1=new Student(3423433,'Dopatka','Frank','Hauptstrasse 4',51580,
                                              'Reichshof','01.10.2002');
  $s2=new Student(8977678,'Grube','Clair','Kanzlergasse 1',16334,
                                                 'Berlin','01.04.2001');
  $p1=new Praktikum('Betriebssysteme','08:00','10:00');
  $p2=new Praktikum('PHP für Anfänger','10:00','12:00');
  $p3=new Praktikum('Netzwerke','14:00','15:00');
  echo 'anmelden Dopatka->Betriebssysteme:<br>';
  echo var_dump($s1->anmelden($p1)).'<br>';
  echo 'anmelden PHP->Dopatka:<br>';
  echo var_dump($p2->anmelden($s1)).'<br>';
  echo 'anmelden Grube->PHP:<br>';
  echo var_dump($p1->anmelden($s2)).'<br>';
  echo 'anmelden Netzwerke->Grube:<br>';
  echo var_dump($s2->anmelden($p3)).'<br>';
  echo '<br><b>Dopatka:</b><br>';
  $data1=$s1->getPraktika();
  foreach ($data1 as $index => $wert){
```

Listing 4.38: Test der Klassen „Student" und „Praktikum"

```
    echo '   '.$wert->getName();
  }
  echo '<br><b>Grube:</b><br>';
  $data2=$s2->getPraktika();
  foreach ($data2 as $index => $wert){
    echo '   '.$wert->getName();
  }
  echo '<br><br><b>Betriebssysteme:</b><br>';
  $data3=$p1->getStudenten();
  foreach ($data3 as $index => $wert){
    echo '   '.$wert->getName();
  }
  echo '<br><b>PHP:</b><br>';
  $data4=$p2->getStudenten();
  foreach ($data4 as $index => $wert){
    echo '   '.$wert->getName();
  }
  echo '<br><b>Netzwerke:</b><br>';
  $data5=$p3->getStudenten();
  foreach ($data5 as $index => $wert){
    echo '   '.$wert->getName();
  }
?>
</body></html>
```

Listing 4.38: Test der Klassen „Student" und „Praktikum" (Forts.)

Nach den Anmeldungen geben zunächst die Studenten ihre Praktika aus und im Anschluss daran die Praktika ihre Teilnehmer. Die folgende Ausgabe zeigt die korrekte Funktionsweise:

anmelden Dopatka->Betriebssysteme:

bool(true)

anmelden PHP->Dopatka:

bool(true)

anmelden Grube->PHP:

bool(true)

anmelden Netzwerke->Grube:

bool(true)

Dopatka:

Betriebssysteme   PHP für Anfänger

Grube:

Betriebssysteme Netzwerke

Betriebssysteme:

Dopatka Grube

PHP:

Dopatka

Netzwerke:

Grube

4.2.4 Komposition und Aggregation

Die Klassen *Student* und *Praktikum* aus dem letzten Kapitel waren relativ unabhängig voneinander. Eine Aggregation bzw. eine Komposition realisiert eine „Besteht aus"-Beziehung. Wenn ein Objekt aus anderen Objekten zusammengesetzt ist, sorgt dies für eine höhere Abhängigkeit als eine „Kennt"-Beziehung. Lediglich die „Ist ein"-Beziehung der Vererbung bindet zwei Klassen noch stärker aneinander.

In den vorherigen Kapiteln wurden bereits die Klassen *Person*, *Kunde*, *Artikel* und *Rechnungsposition* vorgestellt. In diesem Kapitel werden die Klassen nun über die neue Klasse *Rechnung* miteinander verbunden. Eine Rechnung hat eine Beziehung zu genau einem Kunden. Ein Kunde soll seinerseits alle seine Rechnungen kennen.

Eine Rechnung besteht aus Rechnungspositionen, mindestens aus einer Position. Eine Rechnungsposition ohne eine Rechnung macht keinen Sinn. Außerdem gehört eine Rechnungspositionen zu genau einer Rechnung. Hier ist also eine Komposition zu realisieren (Kap. 3.2.2). Eine Rechnungsposition braucht jedoch nicht die Rechnung zu kennen, zu der sie gehört. Die Positionen werden von der zugehörigen Rechnung zentral verwaltet. Daher kann die Klasse *Rechnungspositionen* aus Listing 4.33 unverändert übernommen werden. Die Kundenklasse ist um eine Liste der Rechnungen zu erweitern und zusätzlich ist die Rechnungsklasse zu erstellen. Abbildung 4.10 zeigt das zugehörige Klassendiagramm.

Im ersten Schritt wird die Kundenklasse dahingehend erweitert, dass jeder Kunde eine Liste seiner Rechnungen verwalten kann. Wie schon bei den Studenten und Praktika kommt hier ein Datenfeld zum Einsatz, das hier *$rechnungen* genannt wird. Zusätzlich existiert eine Zählvariable der Rechnungen als interne Eigenschaft der Klasse, die *$anzRechnungen* genannt wird.

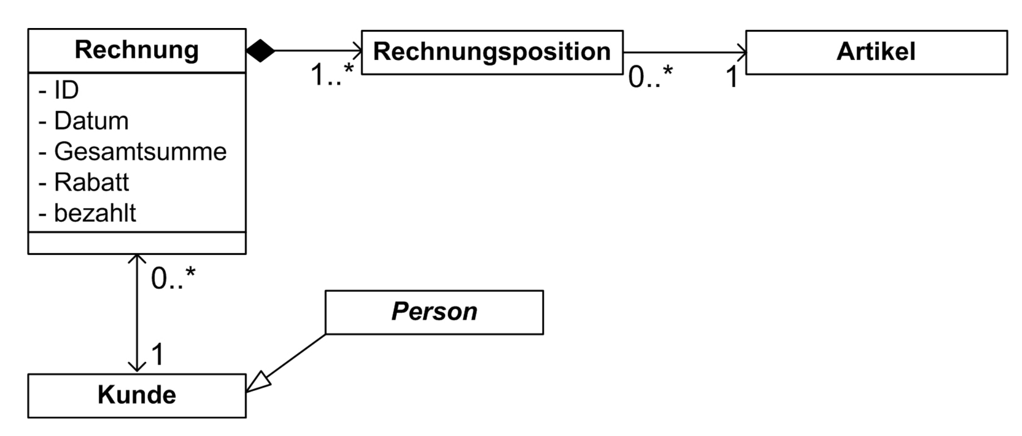


Abbildung 4.10: Klassendiagramm mit zu realisierender Komposition

Eine zusätzliche Get-Methode gibt das Datenfeld der Rechnungen des betreffenden Kunden zurück, und mit der Methode *addRechnung($rechnung)* kann dem Kunden ein neues Rechnungsobjekt übergeben werden. Dieses Objekt wird dann an den nächsten freien Platz im Datenfeld angehängt und der Zähler der Rechnungen wird inkrementiert.

```
<?php
  class Kunde extends Person{
    private $id;
    private $rechnungen=Array(); private $anzRechnungen=0;

    public function __construct($id,$name,$vorname,$strasse,$plz,$ort){
      parent::__construct($name,$vorname,$strasse,$plz,$ort);
      $this->id=$id;
    }

    public function getID(){
      return $this->id;
    }
    public function getRechnungen(){
      return $this->rechnungen;
    }

    public function addRechnung($rechnung){
      $this->rechnungen[$this->anzRechnungen]=$rechnung;
      $this->anzRechnungen++;
    }
  }
?>
```

Listing 4.39: Die erweiterte Kundenklasse

Stellen Sie sich vor, dass Sie die alte Kundenklasse aus Listing 4.25 bereits vielfältig im Einsatz haben. Sie können die existierende Kundenklasse durch die neue Klasse aus Listing 4.39 in allen existierenden Projekten ohne negative Auswirkungen ersetzen. Die neuen Methoden der Rechnungsverwaltung eines Kunden werden in alten Projekten nur nicht aufgerufen bzw. verwendet. Die strikte Modularität der objektorientierten Denkweise vereinfacht also die Versionierung und das Aufspielen verbesserter Komponenten.

In Listing 4.40 wird die neue Rechnungsklasse präsentiert. Die Eigenschaften *$id*, *$datum*, *$rabatt* und *$bezahlt* mit ihren Get- und Set-Methoden sind ebenso unkritisch wie die Referenz auf das Kundenobjekt, das der Rechnung in ihrem Konstruktor übergeben und in *$kunde* festgehalten wird.

Die interessante Frage ist, wie die Komposition zwischen der Rechnung und ihren Positionen realisiert wird. Eine Rechnung benötigt mindestens eine Rechnungsposition, um existieren zu können. Die Daten für diese erste Position wird daher im Konstruktor der Rechnung übergeben, die erste Rechnungsposition im Konstruktor erzeugt, in der Liste der Positionen gespeichert und der Zähler der Rechnungspositionen auf *1* gesetzt.

Zusätzlich existiert in der Rechnungs-Klasse eine Methode *addPosition(...)*, die aber keine zuvor erstellte Rechnungsposition erhält, sondern alle notwendigen Daten zur Erzeugung einer neuen Rechnungsposition. Denn ansonsten müsste der Aufrufer der Methode im Vorfeld eine Rechnungsposition erzeugen, die noch nicht zu einer Rechnung gehört. Dies widerspricht streng betrachtet der Komposition, bei der das Teil (die Position) nicht ohne ihr Ganzes (die Rechnung) existieren kann.

Auch in der Methode *getPositionen()* sollte man bei der Komposition nicht einfach über *return $this->positionen;* das Datenfeld mit den Referenzen auf die Positionsobjekte übergeben. Denn nach einem Destruktor-Aufruf der Rechnung könnten die Positionen oder Kopien via *clone*-Befehl weiter existieren. Die Daten der Rechnungspositionen werden daher in ein Datenfeld umgewandelt und zurückgegeben. Der Aufrufer der Methode kann dann dieses Datenfeld weiterverwenden.

```
<?php
  class Rechnung{
    private $id; private $datum; private $kunde; private $rabatt; // in %
    private $bezahlt=FALSE;
    private $positionen=Array(); private $anzPositionen=0;

    public function __construct($id,$datum,$kunde,$rabattGesamt,
                                        $artikel,$menge,$rabattPos1){
      $this->id=$id; $this->datum=$datum; $this->kunde=$kunde;
      $this->rabatt=$rabattGesamt;
      $this->positionen[0]=new Rechnungsposition($artikel,
                                                  $menge,$rabattPos1);
      $this->anzPositionen=1;
      $kunde->addRechnung($this);
```

Listing 4.40: Die neue Rechnungsklasse

```
    }

    public function getID(){
      return $this->id;
    }
    public function getDatum(){
      return $this->datum;
    }
    public function getKunde(){
      return $this->kunde;
    }
    public function getRabatt(){
      return $this->rabatt;
    }
    public function getBezahlt(){
      return $this->bezahlt;
    }
    public function getGesamtsumme(){
      $summe=0.0;
      foreach ($this->positionen as $index => $wert){
        $summe+=($wert->getGesamtpreis());
      }
      $summe=$summe*(1-($this->rabatt)/100);
      return $summe;
    }
    public function getPositionen(){
      $i=0; $position=Array();
      foreach ($this->positionen as $index => $p){
        $position[$i][0]=$p->getArtikel();
        $position[$i][1]=$p->getMenge();
        $position[$i][2]=$p->getRabatt();
        $position[$i][3]=$p->getEP();
        $position[$i][4]=$p->getGesamtpreis();
        $i++;
      }
      return $position;
    }

    public function setBezahlt($bezahlt){
      if ($bezahlt==TRUE){
```

Listing 4.40: Die neue Rechnungsklasse (Forts.)

```
        $this->bezahlt=TRUE;
      }
      else{
        $this->bezahlt=FALSE;
      }
    }

    public function addPosition($artikel,$menge,$rabattPos){
      $this->positionen[$this->anzPositionen]=new
                        Rechnungsposition($artikel,$menge,$rabattPos);
      $this->anzPositionen++;
    }
  }

?>
```

Listing 4.40: Die neue Rechnungsklasse (Forts.)

Um die Rechnungsklasse zu testen, werden in Listing 4.41 zunächst drei Artikel und ein Kunde angelegt. Im nächsten Schritt wird eine neue Rechnung für diesen Kunden erstellt. Dieser Rechnung werden neben der Rechnungsnummer, dem Rechnungsdatum und der Referenz auf das betreffende Kundenobjekt auch alle Daten übergeben, die zum Erzeugen der ersten Rechnungsposition notwendig sind.

Auf diese Weise wird unmittelbar eine gültige Rechnung erzeugt. Diese Rechnung erhält noch zwei weitere Rechnungspositionen. Das Erzeugen der Rechnung wird dem Kunden im Konstruktor der Rechnung mitgeteilt. Daher können Sie im Anschluss daran das Kundenobjekt direkt nach der Ausgabe seiner Rechnungen durch Aufruf der Methode *getRechnungen()* fragen.

```
<?php require_once("classloader.inc.php"); ?>
<html><body>
<?php
  // definieren
  $a1=new Artikel(1,'Milch','saure Milch',1.00,1.20,7);
  $a2=new Artikel(2,'Butter','gute Butter',0.50,0.70,7);
  $a3=new Artikel(3,'Zucker','süsser Zucker',0.80,1.00,7);
  $k1=new Kunde(1,'Dopatka','Frank','Hauptstrasse 4',51580,'Reichshof');
  // zuweisen
  $r1=new Rechnung(1,'08.10.2009',$k1,5,$a1,3,0);
  $r1->addPosition($a3,1,0);
  $r1->addPosition($a2,7,0);
```

Listing 4.41: Test des Klassengeflechts

```
  // wieder auslesen aus den Objekten
  $rechnungen=$k1->getRechnungen();
  echo '<b>RECHNUNG Nr. '.$rechnungen[0]->getID().' vom '.$rechnungen[0]
                                              ->getDatum().':</b><br>';
  echo 'Kunde: '.$rechnungen[0]->getKunde()->getName().'<br>';
  echo 'Rabatt auf die Gesamtrechnung: '.$rechnungen[0]
                                                ->getRabatt().'%<br>';
  $positionen=$rechnungen[0]->getPositionen();
  $i=0;
  foreach ($positionen as $index => $p){
    echo '<i>Position '.($i+1).':</i><br>';
    echo $p[0]->getName().', '.$p[1].' Stück, '.$p[2].'% Rabatt<br>';
    echo number_format($p[3],2).'EUR pro Stück, '.number_format($p[4],2).
                                                   'EUR gesamt incl. ';
    echo $p[0]->getMwSt().'% MwSt.<br>';
    $i++;
  }
  echo '<b>GESAMTPREIS: '.number_format($rechnungen[0]
                                      ->getGesamtsumme(),2).'EUR';
?>
</body></html>
```

Listing 4.41: Test des Klassengeflechts (Forts.)

Der letzte Teil des Testprogramms besteht darin, die erste Rechnung des Kunden auszugeben. Dabei werden zunächst allgemeine Daten der Rechnung ermittelt, wie die Rechnungsnummer, das Rechnungsdatum, den Namen des Kunden der Rechung sowie den ggf. existierenden Rabatt auf die Gesamtrechnung.

Im Anschluss daran werden alle Rechnungspositionen als Datenfeld ausgelesen und mit der *foreach*-Schleife durchlaufen. Dort werden dann die Daten jeder Rechnungsposition ausgegeben, wie die Nummer der Position, der Name des Artikels, die eingekaufte Menge, der ggf. gewährte Rabatt für diese Position, der Einzelpreis, den Gesamtpreis sowie der Mehrwertsteuersatz für diese Rechnungsposition, wobei der ermäßigte Steuersatz unter anderem für Lebensmittel und Bücher in Deutschland momentan 7 % beträgt und der Steuersatz für andere Waren und Dienstleistungen 19 %.

Nachdem alle Rechnungspositionen in dieser Art ausgegeben wurden, wird abschließend der Gesamtpreis der Rechnung unter Berücksichtigung der Rabatte und der Mehrwertsteuer ausgegeben. Dies führt zur folgenden Ausgabe, die natürlich noch unter Verwendung von HTML-Tabellen und CSS-Formatierungen verschönert werden kann:

RECHNUNG Nr. 1 vom 08.10.2009:

Kunde: Dopatka

Rabatt auf die Gesamtrechnung: 5%

Position 1:

Milch, 3 Stück, 0% Rabatt

1.20EUR pro Stück, 3.85EUR gesamt incl. 7% MwSt.

Position 2:

Zucker, 1 Stück, 0% Rabatt

1.00EUR pro Stück, 1.07EUR gesamt incl. 7% MwSt.

Position 3:

Butter, 7 Stück, 0% Rabatt

0.70EUR pro Stück, 5.24EUR gesamt incl. 7% MwSt.

GESAMTPREIS: 9.66EUR

Sie erkennen an diesem Beispiel, wie hochgradig modular die objektorientierte Entwicklung ablaufen kann. Änderungen sollen dabei nur einen geringen Teil des Quellcodes betreffen. Jede Art von mehrfachem, identischem Quellcode soll durch die Bildung von Ober-Klassen verhindert werden, wie Sie an der Personenklasse erkennen können.

In diesem Beispiel werden die Eingabewerte in die Methoden noch nicht auf Gültigkeit geprüft. Hier sind noch Plausibilitätskontrollen mit einem entsprechenden Fehlermanagement durchzuführen, siehe Kapitel 4.3.

In diesem Beispiel wurde der Fokus auf die Realisierung der Komposition zwischen der Rechnung und ihren Rechnungspositionen gelegt. Die Referenzen auf die Positionen werden von der Rechnungsklasse verwaltet. Sie bleiben innerhalb dieser Klasse und werden nicht nach außen weitergegeben.

Das nächste Beispiel realisiert eine Aggregation. Dazu wurde im dritten Kapitel in der objektorientierten Analyse bereits ein Beispiel aufgezeichnet (Abb. 3.48) und ein Objektdiagramm erstellt (Abb. 3.49). Daraus wurde in Abbildung 3.54 das Klassendiagramm abgeleitet, das nochmals in Abbildung 4.11 dargestellt wird.



Abbildung 4.11: Klassendiagramm mit zu realisierender Aggregation

Die Teile (hier: die Punkte) können im Gegensatz zu einer Komposition auch ohne das Ganze (hier: das Dreieck) existieren. Ein Punkt kann auch zu mehreren Dreiecken gehören. Ein Dreieck besteht jedoch stets aus genau drei Punkten.

Listing 4.42 zeigt die Punktklasse, die keine besonderen Merkmale aufweist. Die beiden Eigenschaften *x* und *y* werden im Konstruktor übergeben und können durch Get-Methoden ausgelesen werden. Die Daten des Punktes können zusätzlich durch die implementierte *__toString*-Methode ausgegeben werden.

```
<?php
  class Punkt{
    private $x; private $y;

    public function __construct($x,$y){
      $this->x=$x; $this->y=$y;
    }

    public function getX(){
      return $this->x;
    }
    public function getY(){
      return $this->y;
    }

    public function __toString(){
      return 'x:'.$this->x.',y:'.$this->y;
    }
  }
?>
```

Listing 4.42: Die Punktklasse

Wichtiger ist die Frage, wie die Aggregation zum Dreieck realisiert wird. Da ein Dreieck genau aus drei Punkten bestehen muss, werden diese Punkte im Konstruktor des Dreiecks übergeben und als Eigenschaften im Dreieck festgehalten. Das Dreieck „besteht" also aus den drei Punkten.

Hier ist als Nächstes die Frage zu stellen, ob man aus beliebigen drei Punkten ein Dreieck bilden kann? Die Antwort lautet nach kurzem Nachdenken: Nein! Wann genau kann man aber kein Dreieck aus drei Punkten bilden?

- Zwei oder drei Punkte verfügen über dieselben x- und y-Koordinaten, sind also inhaltlich gleich. Sie liegen dann übereinander.
- Alle drei Punkte haben dieselbe x- oder dieselbe y-Koordinate. Sie liegen dann parallel zur x-Achse oder zur y-Achse.

Diese ersten beiden Kriterien sind relativ leicht zu entdecken. Viele Programmierer übersehen jedoch, dass auch aus den Punkten P4=(1/1), P5=(2/2) und P6=(3/3) kein Dreieck gebildet werden kann, obwohl die Punkte ungleich sind und keine identischen x- oder y-Koordinaten besitzen. Es kommt also die folgende dritte Bedingung hinzu:

- Drei Punkte bilden kein Dreieck, wenn Sie auf einer Geraden liegen.

Profitipp

Streng genommen müssen Sie im Konstruktor des Dreiecks prüfen, ob das Dreieck überhaupt gebildet werden darf. Die eben geleisteten Vorüberlegungen dazu sind zwingend notwendig, wenn Sie eine langfristig stabile Anwendung bauen wollen. Solche Vorüberlegungen werden jedoch oft aus Zeitmangel nicht durchgeführt oder sie werden als „trivial" abgetan. Daraus resultiert dann eine Anwendung, die in 99.9 % der Fälle korrekt funktioniert, jedoch „plötzlich" völlig falsche Werte liefert.

Im Konstruktor des Dreiecks in Listing 4.43 werden diese Prüfungen durchgeführt in der Hoffnung, keine Spezialfälle übersehen zu haben. Sie erkennen, wie aufwändig dadurch der Konstruktor wird, um ein „einfaches Dreieck" zu erzeugen. Zunächst werden die Punkte wechselseitig auf Gleichheit geprüft sowie auf Parallelität zur x- und zur y-Achse.

Dann wird die Geradengleichung $y=mx+n$ aus den Punkten 1 und 2 gebildet und abschließend geprüft, ob der dritte Punkt auf der Geraden liegt.

Was soll jedoch geschehen, wenn eine der Prüfungen fehlerhaft ist? In diesem Fall darf das Dreieck nicht gebildet werden. Die Rückgabe eines Konstruktor-Aufrufs ist jedoch immer eine Objektreferenz, ein *FALSE* als Rückgabe im Konstruktor ist nicht erlaubt. Zum jetzigen Zeitpunkt besteht nur die Möglichkeit, das Skript hart über den *die*-Befehl in einer separaten internen Fehlermethode abzubrechen. In Kapitel 4.3 wird jedoch ein objektorientiertes Konzept der Fehlerbehandlung vorgestellt, bei dem der Konstruktoraufruf ein Fehlerobjekt zurückgeben kann, das der Aufrufer dann auswerten kann bzw. muss.

```
<?php
  class Dreieck{
    private $p1; private $p2; private $p3;

    public function __construct($p1,$p2,$p3){
      $this->p1=$p1; $this->p2=$p2; $this->p3=$p3;
      // Gleichheit ist verboten
      if ($p1->getX()==$p2->getX()){
        if ($p1->getY()==$p2->getY()) $this->fehler();
      }
      if ($p1->getX()==$p3->getX()){
        if ($p1->getY()==$p3->getY()) $this->fehler();
      }
      if ($p2->getX()==$p3->getX()){
        if ($p2->getY()==$p3->getY()) $this->fehler();
      }
      if (($p1->getX()==$p2->getX())&&($p2->getX()==$p3->getX()))
```

Listing 4.43: Die Dreieckklasse

```
                                                    $this->fehler();
      if (($p1->getY()==$p2->getY())&&($p2->getY()==$p3->getY()))
                                                    $this->fehler();
      // y=m*x+n
      // m=(y2-y1)/(x2-x1)
      $m=(($p2->getY())-($p1->getY()))/(($p2->getX())-($p1->getX()));
      // n1=n2=y2-m*x2
      $n=($p1->getY())-$m*($p1->getX());
      $y_gerade=$m*($p3->getX())+$n;
      if ($y_gerade==($p3->getY())) $this->fehler();
    }

    private function fehler(){
      die('Aus diesen 3 Punkten kann kein Dreieck gebildet werden!');
    }

    public function __toString(){
      return 'P1('.$this->p1.');P2('.$this->p2.');P3('.$this->p3.')';
    }
  }
?>
```

Listing 4.43: Die Dreieckklasse (Forts.)

Die beiden neu erstellten Klassen werden – wie immer – getestet. Dabei werden im Hauptprogramm zunächst drei Punkte mit ihren Koordinaten erzeugt. Die Punkte können ja auch ohne Dreiecke existieren. Dann wird aus diesen drei Punkten ein Dreieck erzeugt und dessen Daten werden ausgegeben.

Der zweite Test erzeugt drei neue Punkte, die jedoch auf einer Geraden liegen. Hier sollte die Erzeugung des Dreiecks nicht erfolgreich sein.

```
<?php require_once("classloader.inc.php"); ?>
<html><body>
<?php
  $p1=new Punkt(0,10); $p2=new Punkt(5,5); $p3=new Punkt(10,5);
  $d1=new Dreieck($p1,$p2,$p3);
  echo $d1.'<br>';
  $p4=new Punkt(1,1); $p5=new Punkt(2,2); $p6=new Punkt(3,3);
  $d2=new Dreieck($p4,$p5,$p6);
  echo $d2.'<br>';
?>
</body></html>
```

Listing 4.44: Erstellung von Punkten und Dreiecken

Die Ausgabe des Tests erfolgt wie erwartet:

P1(x:0,y:10);P2(x:5,y:5);P3(x:10,y:5)

Aus diesen 3 Punkten kann kein Dreieck gebildet werden!

4.2.5 Interfaces und deren Implementierung

Bereits in der Definition der objektorientierten Grundlagen wurde der Sinn von Interfaces begründet. Sie stellen das Prinzip der Datenkapselung der Objektorientierung im höchsten Maße dar. Abbildung 3.67 zeigt ein in UML definiertes Interface eines Datenzugriffs jeglicher Art. Dabei kann lesend und schreibend zugegriffen werden

- auf eine Datei
- auf eine Datenbank
- auf eine Netzwerkverbindung

Es stellt sich die Frage, welche Funktionalität (also welche Methoden) für einen allgemeinen Datenzugriff notwendig ist. Dies ist

- das Öffnen der Datenquelle unter Angabe von Parametern wie der Pfad zur Datenquelle und/oder Daten zur Authentifizierung an der Datenquelle
- das Schließen einer geöffneten Datenquelle
- das Schreiben in die Datenquelle unter Angabe der zu schreibenden Daten und ggf. unter Angabe der Parameter, die für den Schreibvorgang notwendig sind
- das Lesen aus einer Datenquelle unter Angabe der Parameter für den Lesezugriff, wobei die ausgelesenen Daten zurückgegeben werden

Listing 4.45 zeigt die Definition des Interfaces in PHP5. Von Interesse sind insbesondere die übergebenen Parameter *$param*. Wie lässt sich eine allgemeine Definition finden, wie man Parameter übergibt? Diese Definition muss sowohl für Dateien und Datenbanken als auch für Netzwerkzugriffe gültig sein.

```
<?php
  interface iDZ{
    public function öffnen($param); public function schliessen();
    public function lesen($param); public function schreiben($param);
  }
?>
```

Listing 4.45: Das Datenzugriffsinterface iDZ

Listing 4.46 zeigt eine Hilfskasse *ParameterListe*, die dynamische Parameterobjekte in einem assoziativen Datenfeld erzeugt. Die Idee besteht darin, dass der Anwender des Interfaces ein Parameterobjekt erzeugt und es mit Parametern über die *add*-Methode füllt.

Die Dokumentation einer konkreten Implementierung des Datenzugriffsinterfaces *iDZ* muss die Namen und die möglichen Werte der übergebenen Parameter für jede Methode enthalten.

Neben der *add*-Methode können mit der Hilfskasse noch bestehende Parameter anhand ihres Namens ausgelesen und die Existenz eines Parameternamens abgefragt werden. Die *kill*-Methode setzt das gesamte Parameterobjekt zurück, sodass Sie es für eine weitere Übergabe verwenden können.

```
<?php
  class ParameterListe{
    private $paramListe=Array();

    public function add($name,$wert){
      $this->paramListe[$name]=$wert;
    }
    public function get($name){
      return $this->paramListe[$name];
    }
    public function isParam($name){
      return isset($this->paramListe[$name]);
    }
    public function kill(){
      unset($this->paramListe);
    }
  }
?>
```

Listing 4.46: Die Hilfsklasse für die Parameterübergabe

Im nächsten Schritt wird das Interface *iDZ* exemplarisch realisiert für den Zugriff auf eine MySQL-Datenbank. Tabelle 2.28 im zweiten Kapitel zeigte bereits die notwendigen Funktionen, um auf eine MySQL-Datenbank zuzugreifen.

Die erste implementierte Funktion ist das Öffnen der Datenquelle. MySQL verlangt dabei vier Parameter, nämlich

- die IP-Adresse des Hosts, auf dem der MySQL-Datenbankserver installiert ist
- den Benutzernamen und das Passwort zur Authentifikation am Datenbankserver
- die zu öffnende Datenbank

Der erste Schritt prüft das übergebene Parameterobjekt auf Gültigkeit. Im Anschluss daran wird die Verbindung zum Datenbankserver aufgebaut und versucht, die angegebene Datenbank zu öffnen. Die Referenz auf die geöffnete Verbindung wird in der Eigenschaft *$conn* festgehalten. Je nach Erfolg gibt die *öffnen*-Methode *TRUE* oder *FALSE* zurück.

Die *schließen*-Methode aus dem Interface ist durch den Befehl *mysql_close()* leicht zu implementieren.

Die Prüfung der Eingabeparameter erfolgt auch von der *lesen*- und *schreiben*-Methode. Beide Methoden verlangen einen SQL-Befehl im Parameter *$sql* des Parameterobjekts. Beim Lesen aus der Datenquelle gibt der Befehl *mysql_query($sql)* ein Resultset in Form einer Tabelle in der Variablen *$data* zurück.

Diese Tabelle wird umgewandelt in ein Datenfeld, das aus Datensätzen besteht. Jeder Datensatz ist selbst ein Datenfeld, das die Daten aus der Datenquelle enthält. Die Tabelle wird also in ein zweidimensionales Datenfeld umgewandelt, dessen Referenz *$ausgabe* zurückgegeben wird.

Die Methode *schreiben* funktioniert auf ähnliche Weise. Hier gibt der Befehl *mysql_query($sql)* jedoch entweder *TRUE* oder *FALSE* zurück, je nachdem, ob der SQL-Befehl erfolgreich ausgeführt wurde oder nicht. Diese Ausgabe wird direkt als Ausgabe der Methode zurückgegeben.

Die interne Hilfsmethode *starts_with* gibt zurück, ob ein Text mit einer bestimmten Zeichenfolge beginnt, wobei nicht zwischen Groß- und Kleinschreibung unterschieden wird. So werden die Befehle der übergebenen SQL-Anweisung geprüft, die in Form einer Zeichenkette übergeben wird.

```
<?php
  class mysqlDZ implements iDZ{
    private $conn;

    public function öffnen($p){
      if (!isset($p)) return FALSE;
      if ((!$p->isParam('host'))||(!$p->isParam('user'))||(!$p
                               ->isParam('pass'))||(!$p->isParam('db'))){
        return FALSE;
      }
      $host=$p->get('host'); $user=$p->get('user');
      $pass=$p->get('pass'); $db=$p->get('db');
      $conn=@mysql_connect($host,$user,$pass);
      if ($conn){
        if (@mysql_select_db($db,$conn)==1){
          return TRUE;
        }
        else{
          return FALSE;
        }
      }
      else{
```

Listing 4.47: Die Implementierung des Interfaces für einen MySQL-Zugriff

```
        return FALSE;
      }
    }

    public function schliessen(){
      @mysql_close();
    }

    public function lesen($p){
      if (!isset($p)) return FALSE;
      if (!$p->isParam('sql')) return FALSE;
      $sql=$p->get('sql');
      if ($this->starts_with($sql,'SELECT')==FALSE) return FALSE;
      $data=@mysql_query($sql);
      if ($data==FALSE) return (FALSE);
      $ausgabe=Array();
      $x=0;
      while($row=mysql_fetch_row($data)){
        $datensatz=Array();
        for($i=0;$i<count($row);$i++){
          $datensatz[$i]=$row[$i];
        }
        $ausgabe[$x]=$datensatz;
        $x++;
      }
      return $ausgabe;
    }

    public function schreiben($p){
      if (!isset($p)) return FALSE;
      if (!$p->isParam('sql')) return FALSE;
      $sql=$p->get('sql');
      if (($this->starts_with($sql,'UPDATE')==TRUE)||($this
                               ->starts_with($sql,'INSERT')==TRUE)){
        // UPDATE oder INSERT
        return (@mysql_query($sql));
      }
      else{
        return FALSE; // FEHLER
      }
```

Listing 4.47: Die Implementierung des Interfaces für einen MySQL-Zugriff (Forts.)

```
    }

    private function starts_with($str,$wert){
      return strtolower(substr($str,0,strlen($wert)))==strtolower($wert);
    }
  }
?>
```

Listing 4.47: Die Implementierung des Interfaces für einen MySQL-Zugriff (Forts.)

Der Datenzugriff kann natürlich um Transaktionen und/oder verschlüsselten Zugriff ergänzt werden. Es wurde im UML-Teil des dritten Kapitels bereits erklärt, dass Interfaces auch vererbt werden können, um zusätzliche Funktionalität hinzuzufügen. Dies kann beispielsweise durch die Definition *interface iCryptedDZ extends iDZ* erfolgen.

Nun muss der Zugriff auf diese Implementierung noch getestet werden. Dazu wird auf die existierende Börsendatenbank aus dem zweiten Kapitel zurückgegriffen (Abb. 2.11 ff.). In der Datenbank *boerse* ist eine Tabelle *ag* enthalten, die einen Identifikator und den Namen von Aktiengesellschaften enthält. Zum Testen wird die Verbindung zum Datenbankserver geöffnet, der Name einer Aktiengesellschaft aktualisiert, anschließend die gesamte Tabelle ausgelesen und im letzten Schritt wird die Verbindung wieder geschlossen.

Im Gegensatz zum zweiten Kapitel wird die Datenbankverbindung hier innerhalb des Verbindungsobjekts *$db* verwaltet. Dieses Objekt verfügt durch die Implementierung des zuvor definierten Interfaces über die Methoden

- öffnen
- schreiben
- lesen
- schließen

Für die Parametrierung werden beim Öffnen, Schreiben und Lesen eigene Parameterobjekte definiert, die beim Öffnen über die Parameter *host*, *user*, *pass* und *db* sowie beim Schreiben und Lesen über den Parameter *sql* verfügen.

Erkennen Sie den Mehrwert gegenüber der Realisierung im zweiten Kapitel? Der Programmierer, der ein *mysqlDZ*-Objekt verwendet, muss nichts über die MySQL-Befehle von PHP wissen. Er muss lediglich die zum Öffnen notwendigen Parameter kennen und die SQL-Sprache beherrschen. Über dasselbe Interface könnten Sie auch Implementierungen für eine Oracle-, MS-SQL- oder eine DB2-Datenbank schreiben. Die Verwendung wäre identisch. Gegebenenfalls müssten die Parameter etwas verändert werden. Mit ähnlichen Parametern lassen sich auf die gleiche Art und Weise auch Zugriffe auf Dateien realisieren.

Die Eingabe der Parameter erfolgt jedoch üblicherweise über eine Konfigurationseingabemaske, die einem eingeloggten Administrator zur Verfügung steht. Für jede Imple-

mentierung muss also im Frontend noch eine passende Eingabemaske erstellt werden, mit der die Parameter festgelegt werden.

Doch zunächst zurück zum Testprogramm. Bei *$p_öffnen*, *$p_schreiben* und *$p_lesen* handelt es sich um die Parameterobjekte, die gemäß den Vorgaben aus der Implementierung *mysqlDZ* gefüllt werden. Das Verbindungsobjekt selbst heißt *$db*. Der schreibende Zugriff *$db->schreiben($p_schreiben)* liefert als Ergebnis lediglich einen Wahrheitswert, der den Erfolg des Schreibzugriffs widerspiegelt.

Interessant ist der lesende Zugriff. Wie bereits beschrieben wurde, liefert *$ausgabe=$db->lesen($p_lesen)* ein zweidimensionales Feld als Ergebnismenge in der Referenz *$ausgabe* zurück. Mit den PHP-Befehlen *count($ausgabe)* können Sie die Anzahl der Datensätze ermitteln und mit *count($ausgabe[0])* die Anzahl der zurückgegebenen Spalten der Ergebnistabelle. Das sind die Spalten, die Sie hinter dem SELECT-Befehl der SQL-Anweisung angegeben haben. Mit der verschachtelten *foreach*-Schleife können Sie nun auf jedes Datenelement zugreifen.

```
<?php require_once("classloader.inc.php"); ?>
<html><body>
<?php
  // neues DB-Verbindungsobjekt erstellen, welches das Interface iDZ
  // implementiert
  $db=new mysqlDZ();
  // 1. öffnen
  $p_öffnen=new ParameterListe();
  $p_öffnen->add('host','localhost'); $p_öffnen->add('user','root');
  $p_öffnen->add('pass',''); $p_öffnen->add('db','boerse');
  if ($db->öffnen($p_öffnen)==FALSE) die('FEHLER beim DB-Öffnen!');
  // 2. schreiben:
  $p_schreiben=new ParameterListe();
  $p_schreiben->add('sql','UPDATE ag SET name="Dopatka AG" WHERE ID=6');
  echo 'Schreiben erfolgreich:';
  echo var_dump($db->schreiben($p_schreiben)); echo '<br>';
  // 3. lesen:
  $p_lesen=new ParameterListe();
  $p_lesen->add('sql','SELECT ID,name FROM ag ORDER BY ID');
  $ausgabe=$db->lesen($p_lesen);
  if ($ausgabe==FALSE){
    $db->schliessen();
    die('FEHLER beim DB-Zugriff!');
  }
  echo 'Anzahl Datensätze:'.count($ausgabe).'<br>';
  echo 'Anzahl Spalten:'.count($ausgabe[0]).'<br>';
  foreach($ausgabe as $index => $datensatz){
```

Listing 4.48: Test der Implementierung

```
    foreach($datensatz as $index2 => $wert){
      echo $wert;
      echo '<br>';
    }
  }
  // 4. schliessen
  $db->schliessen();
?>
</body></html>
```

Listing 4.48: Test der Implementierung (Forts.)

Auf diese Weise wird die folgende Ausgabe erzeugt. Als Übung können Sie diese Ausgabe in eine HTML-Tabelle umformatieren.

Schreiben erfolgreich:bool(true)

Anzahl Datensätze:30

Anzahl Spalten:2

1

ADIDAS-SALOMON AG

2

ALLIANZ AG VNA O.N

3

ALTANA AG O.N.

4

BASF AG O.N.

5

BMW

6

Dopatka AG

...

4.2.6 Umsetzung von Sequenzdiagrammen

Wie Sie aus einem bestehenden UML-Klassendiagramm eine Klasse in PHP 5 ableiten können, haben Sie bereits in den vorherigen Kapiteln erfahren. Hier wird die Definition der Klassen, der Eigenschaften und Methoden sowie der Beziehungen der Klassen untereinander fokussiert. Ebenso wurde bereits ein Aktivitätsdiagramm aus Abbildung 4.9 in Listing 4.36 in einer Methode umgesetzt.

Die Beziehung zwischen einem UML-Sequenzdiagramm und einer PHP-Klasse wurde jedoch noch nicht vorgestellt. Wie ein Aktivitätsdiagramm zeigt ein Sequenzdiagramm einen Ablauf, der jedoch weniger einen Geschäftsprozess abbildet, sondern eher die Interaktion von Objekten in den Vordergrund stellt.

Der Quellcode aus Listing 4.49 ist ein Ausschnitt aus dem bereits vorgestellten Quellcode des Listings 4.41, der den Test des Klassengeflechts zwischen Kunden, Rechnungen, Rechnungspositionen und Artikeln realisiert. Es wird also eine Kommunikation von Objekten von vier verschiedenen Klassen abgebildet. Dieser existierende Quellcode soll in diesem Beispiel in einem Sequenzdiagramm auf Muschelebene dokumentiert werden.

```
$rechnungen=$k1->getRechnungen();
echo '<b>RECHNUNG Nr. '.$rechnungen[0]->getID().' vom '.$rechnungen[0]
                                              ->getDatum().':</b><br>';
echo 'Kunde: '.$rechnungen[0]->getKunde()->getName().'<br>';
echo 'Rabatt auf die Gesamtrechnung: '.$rechnungen[0]
                                                ->getRabatt().'%<br>';
$positionen=$rechnungen[0]->getPositionen();
$i=0;
foreach ($positionen as $index => $p){
  echo '<i>Position '.($i+1).':</i><br>';
  echo $p[0]->getName().', '.$p[1].' Stück, '.$p[2].'% Rabatt<br>';
  echo number_format($p[3],2).'EUR pro Stück, '.number_format($p[4],2).
                                                    'EUR gesamt incl. ';
  echo $p[0]->getMwSt().'% MwSt.<br>';
  $i++;
}
```

Listing 4.49: Quellcodeausschnitt aus Listing 4.41

Zunächst wird über den Frontend-Quellcode – den man als Akteur gegenüber den anderen Objekten sehen kann – die Methode *getRechnungen* des Kunden *Dopatka* aufgerufen. Daraufhin erhält der Aufrufer eine Liste der Rechnungen als Rückgabe. Von dieser Liste wird das erste Element, *$rechnungen[0]* betrachtet. Dies ist ein Objekt der Klasse *Rechnung*. Von dieser Rechnung werden nun Eigenschaften ausgelesen, nämlich

- die Rechnungsnummer (ID)
- das Rechnungsdatum
- der Name des Kunden der Rechnung
- der Rabatt auf die Gesamtrechnung
- die Anzahl der Rechnungspositionen

Genau dieses Auslesen erkennen Sie im Sequenzdiagramm der Abbildung 4.12. Im Anschluss daran wird jede Rechnungsposition in einer Schleife durchgegangen. Die Schleife kann in einem Sequenzdiagramm nur schwer abgebildet werden und wird in

dem gepunkteten Kasten mit der Beschriftung **positionen* (heißt: für alle Positionen) dargestellt. Für jede Rechnungsposition werden nun ausgegeben:

- der Name des Artikels
- die bestellte Menge
- der Rabatt dieser einzelnen Position
- der Einzelpreis
- der Gesamtpreis dieser Position
- der Mehrwertsteuersatz

Der Name des Artikels wird über die Artikelreferenz ausgelesen, die jede Rechnungsposition besitzt. Im Sequenzdiagramm der Abbildung 4.12 werden nur die ersten beiden Lesevorgänge aus jeder Rechnungsposition, also Artikelname und Menge, dargestellt.

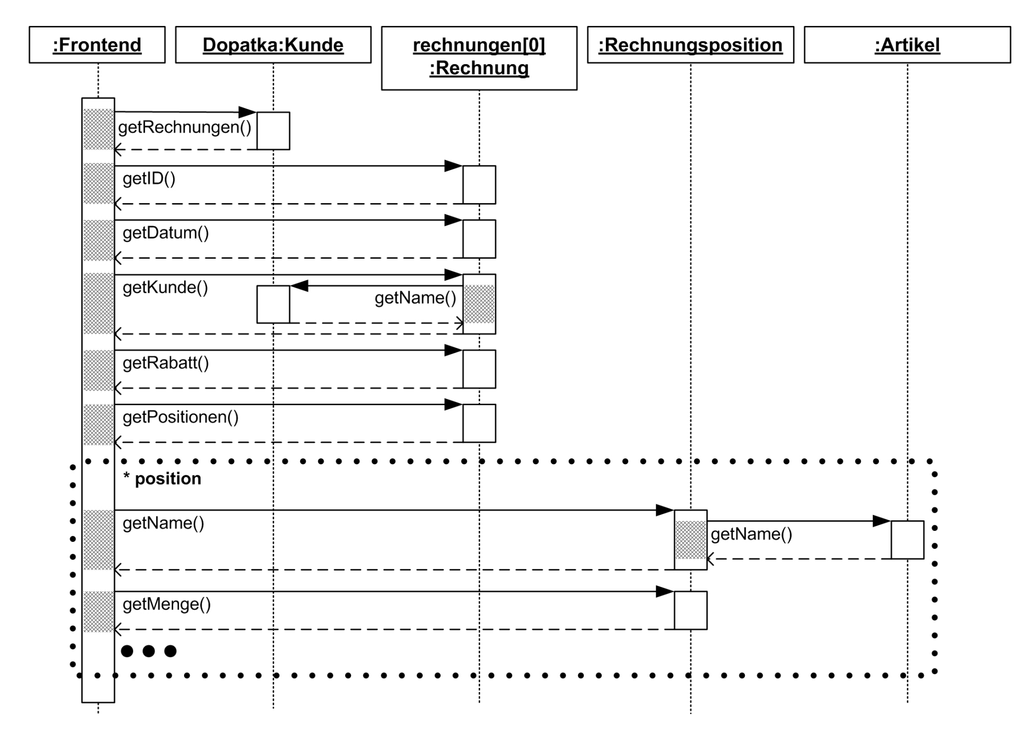


Abbildung 4.12: Sequenzdiagramm des PHP-Codes aus Listing 4.49

4.2.7 Umsetzung von Zustandsautomaten

Auch die Implementierung eines in UML definierten Zustandsdiagramms ist in PHP 5 möglich. Die folgende Abbildung 4.13 wurde bereits im dritten Kapitel zur Beschreibung des Zustandsdiagramms der UML verwendet. Es beschreibt das Interface einer Flugreservierung mit den Methoden

- reservieren
- stornieren
- buchen

Dies ist zunächst Bestandteil eines Klassendiagramms. Zusätzlich wird jedoch ein Protokoll in Form eines Zustandsdiagramms dargestellt. Dieses Protokoll zeigt, in welcher Reihenfolge die Methoden einer Klasse abgearbeitet werden müssen, die das Interface und das Protokoll implementieren soll.

> **Hinweis**
>
> Sie erkennen daran, dass jedes UML-Diagramm der Designphase direkten Einfluss auf den entstehenden Quellcode haben kann. Dies gilt insbesondere für die Fisch- und Muschelebene. Jedes Diagramm stellt einen anderen Aspekt des Quellcodes dar und kann direkt in Quellcode übersetzt werden. Ebenso ist eine Übersetzung von Quellcode in UML zu Zwecken der Dokumentation möglich.

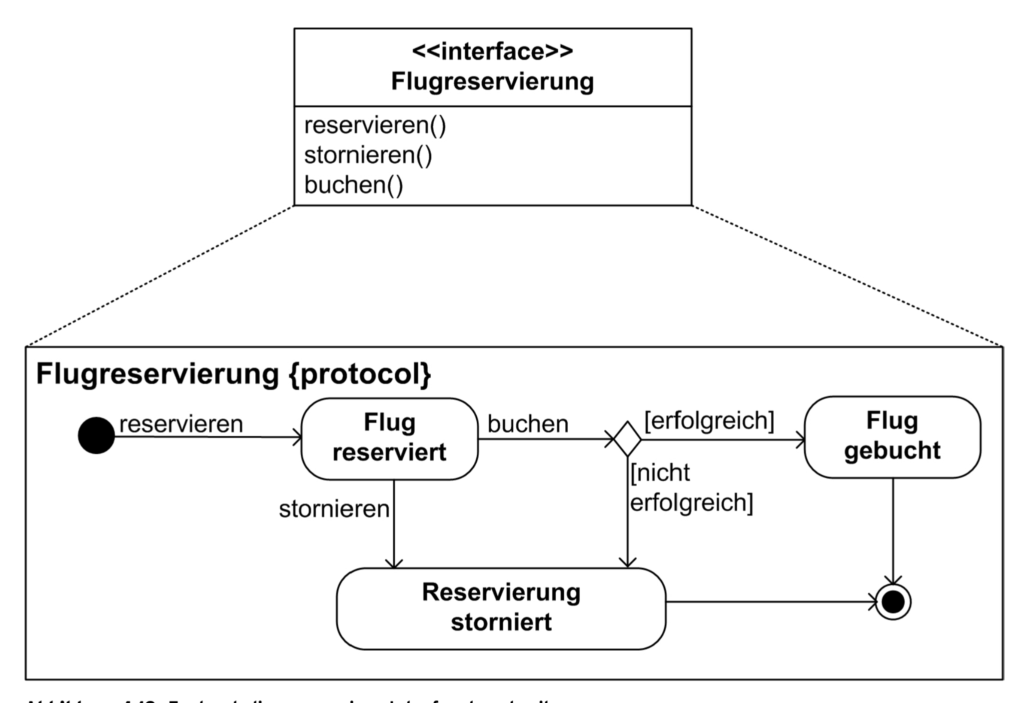


Abbildung 4.13: Zustandsdiagramm einer Interfacebeschreibung

Listing 4.50 zeigt im ersten Schritt die Definition des Interfaces zur Flugreservierung mit den drei Methoden.

```
<?php
  interface iFlugreservierung{
```

Listing 4.50: Das Interface der Flugreservierung

```
    public function reservieren();
    public function stornieren();
    public function buchen();
  }
?>
```

Listing 4.50: Das Interface der Flugreservierung (Forts.)

Listing 4.51 zeigt nun die Implementierung des Interfaces unter Berücksichtigung des im Zustandsdiagramm definierten Protokolls. Insgesamt existieren die Zustände

1. initialisiert
2. reserviert
3. gebucht
4. storniert

Der aktuelle Zustand wird in der Eigenschaft *$zustand* festgehalten und im Konstruktor auf 0 gesetzt. Zusätzlich wird eine Methode *getZustand()* implementiert, mit der man den Zustand des Reservierungsobjekts jederzeit abfragen kann.

Dem folgen die im Interface deklarierten Methoden *reservieren*, *stornieren* und *buchen*. Dabei wird stets im ersten Schritt geprüft, ob die entsprechende Methode im aktuellen Zustand überhaupt ausgeführt werden darf. Ist dies nicht der Fall, so gibt die Methode *FALSE* zurück. Ansonsten erfolgt ein Zustandswechsel und die entsprechende Methode gibt den Wahrheitswert *TRUE* zurück.

Zu berücksichtigen ist noch, dass nicht jede Reservierung in einer erfolgreichen Buchung endet. Dies geschieht dann, wenn die Reservierung zu spät in eine Buchung umgewandelt wird und bereits viele andere Kunden den Flug ausgebucht haben. Da jedoch in diesem Beispiel kein vollständiges Reservierungssystem implementiert werden soll, wird die Reservierung per Zufallssystem in eine Buchung umgewandelt. Dazu wird mit *srand(microtime()*1000000)* ein Zufallszahlengenerator auf Basis der Systemzeit initialisiert. Der Befehl *$zufall=rand(0,2)* gibt dann eine Zufallszahl zwischen 0 und 2 zurück. Ist die erstellte Zahl größer als 1, so gilt die Buchung als erfolgreich, ansonsten nicht. Der Zustandswechsel innerhalb einer Methode wird dann wiederum in der Eigenschaft *$zustand* festgehalten.

```
<?php
  class AirlineReservierung implements iFlugreservierung{
    // 0:init, 1:reserviert, 2:gebucht, 3:storniert
    private $zustand;

    public function __construct(){
      $this->zustand=0;
    }
```

Listing 4.51: Die Interfaceimplementerung und Umsetzung des Zustandsautomaten

```
    public function getZustand(){
      switch($this->zustand){
        case 0:
          return 'initialisiert'; break;
        case 1:
          return 'reserviert'; break;
        case 2:
          return 'gebucht'; break;
        case 3:
          return 'storniert'; break;
        default:
          return 'FEHLER: Ungültiger Zustand!'; break;
      }
    }

    public function reservieren(){
      if ($this->zustand!=0) return FALSE;
      $this->zustand=1; // reserviert
      return TRUE;
    }

    public function stornieren(){
      if ($this->zustand!=1) return FALSE;
      $this->zustand=3; // storniert
      return TRUE;
    }

    public function buchen(){
      if ($this->zustand!=1) return FALSE;
      // es ist Zufall, ob die Buchung funktioniert...
      srand(microtime()*1000000);
      $zufall=rand(0,2);
      if ($zufall>1){
        $this->zustand=3; // gebucht
      }
      else{
        $this->zustand=2; // storniert
      }
      return TRUE;
    }
```

Listing 4.51: Die Interfaceimplementerung und Umsetzung des Zustandsautomaten (Forts.)

```
  }
?>
```

Listing 4.51: Die Interfaceimplementerung und Umsetzung des Zustandsautomaten (Forts.)

Der Test der Implementierung erfolgt, indem ein Objekt der *AirlineReservierung* angelehnt wird. Im Anschluss daran wird ein Pfad im Zustandsdiagramm durchgegangen und nach jedem Schritt der aktuelle Zustand des Objekts ausgegeben.

```
<?php require_once("classloader.inc.php"); ?>
<html><body>
<?php
  $FDairline=new AirlineReservierung();
  echo $FDairline->getZustand().'<br>';
  $FDairline->reservieren();
  echo $FDairline->getZustand().'<br>';
  $FDairline->buchen();
  echo $FDairline->getZustand().'<br>';
?>
</body></html>
```

Listing 4.52: Test der Umsetzung des Zustandsautomaten

Die Ausgabe ist im Folgenden dargestellt, wobei in ca. 50 % der Fälle die dritte Ausgabe *storniert* lautet:

initialisiert

reserviert

gebucht

4.3 Objektorientierte Fehlerbehandlung

Bereits im zweiten Kapitel wurde der @-Operator zur Fehlerunterdrückung vorgestellt. So wird mit *$datei=@fopen("counter.txt","w")* die Meldung *Warning: fopen(counter.txt) [function.fopen]: failed to open stream...* unterdrückt. Über die Prüfung *if($datei===FALSE){...}* könnte dann eine Behandlung des Fehlers erfolgen, wenn die Datei nicht existiert. Auch in Listing 4.51 werden Fehlerprüfungen durchgeführt, indem eine *if*-Verzweigung zum Einsatz kommt und bei einem Fehler der Rückgabewert einer Methode in besonderem Maße erfolgt. Die implementierten Methoden *reservieren*, *stornieren* und *buchen* liefern im Fehlerfall *FALSE* zurück.

Auch die Implementierung der Datenbankzugriffsschnittstelle in Listing 4.48 arbeitet auf diese Weise. Das Lesen aus der Datenbank liefert ein Datenfeld als Rückgabe oder *FALSE*, wenn das Lesen nicht erfolgreich war.

Doch wird diese besondere Fehlerrückgabe vom Aufrufer einer Methode ausgewertet? Die Antwort lautet: Meistens nicht! Dies liegt daran, dass man oft dazu neigt, nur den erfolgreichen Fall zu betrachten, um möglichst schnell eine lauffähige Anwendung zu erhalten.

Im Fall der Flugreservierung hat dies zur Folge, dass Sie unter Umständen einen Flug in einem Zustand stornieren wollen, der ungleich *reserviert* ist. In diesem Fall liefert die Methode *stornieren* den Wert *FALSE*. Wird dieser Rückgabewert nicht ausgewertet, so geht der Aufrufer der Methode davon aus, dass sein Flug erfolgreich storniert wurde. Nun könnte man sagen: Aber dann hätte der Wert doch ausgewertet werden müssen!

Die Verantwortung für die Fehlerbehandlung wird also vom Programmierer der Klasse auf den Aufrufer der Methode weitergegeben. Dieser muss dann nach jedem Methodenaufruf eine Prüfung vornehmen, ob der Aufruf erfolgreich war oder nicht. Dies führt dazu, dass Sie den regulären, erfolgreichen Ablauf des Programms bei vielen Zeilen Quellcode kaum noch nachvollziehen können, da er ständig von der Fehlerprüfung unterbrochen wird.

So zeigt Listing 4.53 einen Ausschnitt aus dem bereits vorgestellten Listing des objektorientierten Datenzugriffs mit dem Datenzugriffsobjekt *$db*. Der reguläre Programmablauf ist fett gedruckt, während der restliche Code zur Fehlerbehandlung dient. Bei größerem Quellcode entstehen viele Fehlerpfade im Quellcode, sodass Sie den regulären Ablauf des Programms kaum noch erkennen können.

```
$p_lesen=new ParameterListe();
$p_lesen->add('sql','SELECT ID,name FROM ag ORDER BY ID');
$ausgabe=$db->lesen($p_lesen);
if ($ausgabe==FALSE){
  $db->schliessen();
  die('FEHLER beim DB-Zugriff!');
}
echo 'Anzahl Datensätze:'.count($ausgabe).'<br>';
echo 'Anzahl Spalten:'.count($ausgabe[0]).'<br>';
```

Listing 4.53: Ausschnitt aus Listing 4.48

Fehler in einer Klasse: werfen und fangen

Das Beispiel in Listing 4.53 zeigt ein Datenbankobjekt, das Fehler produzieren kann. Bei der Flugreservierung in Abbildung 4.13 sind nicht alle Methodenaufrufe in jedem Zustand erlaubt. Die jeweilige Methode des Reservierungsobjekts gibt bei einem ungültigen Aufruf ebenso *FALSE* zurück wie die Methode des Datenbankobjekts.

Mit der fünften Version hat PHP nun ein Konzept zur Fehlerbehandlung eingeführt, das bei anderen objektorientierten Sprachen wie Java, VB.NET oder C# bereits sehr erfolgreich ist. Dabei wird der Quellcode, der Fehler produzieren kann, an einem Stück in einem so genannten *try*-Block ausgeführt. Die Idee ist, dass man zunächst versucht, den Quellcode an einem Stück auszuführen. Ist dies erfolgreich, wird der aufrufende Code linear abgearbeitet. Ansonsten ist ein Fehler aufgetreten, der von einer aufgerufenen

Methode eines Objekts geworfen wurde. Dieser Fehler wird dann in einem separaten Quellcodebereich gefangen und behandelt.

Damit dieses Konzept funktioniert, müssen sich zunächst die Rückgabewerte der Methoden auf ihre eigentliche Funktion besinnen; nämlich Werte, Objektreferenzen oder Datenfelder zurückzugeben. Rückgabewerte geben in diesem Konzept also **keine** Fehlermeldungen wie *FALSE* oder bestimmte Error-Codes zurück. Stattdessen können diese Methoden auf einem neuen, unabhängigen Fehlerkanal ihre Fehlermeldungen zurückgeben.

In der objektorientierten Denkweise sind auch Fehler Objekte, die Eigenschaften und Methoden besitzen. Da Objekte von der Schablone einer Klasse erzeugt werden, können Sie nun eigene Fehlerklassen schreiben und ein Fehlermanagement ihrer komplexen Anwendung einführen.

Listing 4.54 skizziert eine erste Fehlerklasse, die von der in PHP vordefinierten Klasse *Exception* vererbt wird. Damit ist die eigene Klasse eine Exception, die geworfen und behandelt werden kann. Unsere Fehlerklasse besteht aus

- einer Fehlernummer
- einer Fehlermeldung
- einer Eigenschaft, in der man den Namen der Methode speichern kann, in der der Fehler aufgetreten ist
- einem Wahrheitswert, der signalisiert, ob ein Fehler kritisch ist oder nicht

Im Konstruktor werden die vier Eigenschaften wie üblich gesetzt. Beim Erzeugen eines kritischen Fehlerobjekts könnte man beispielsweise automatisch eine E-Mail an den Administrator absetzen (Kap. 2.2) oder ein Logging in eine Datenbank und/oder in eine Textdatei vornehmen.

Zusätzlich besitzt die eigene Fehlerklasse Get-Methoden, um die gesetzten Eigenschaften auszulesen sowie eine *toString()*-Methode, damit der Fehler unmittelbar vorformatiert ausgegeben werden kann.

```
<?php
  class Fehler extends Exception{
    private $nummer; private $meldung;
    private $methode; private $kritisch=FALSE;

    public function __construct($nummer,$meldung,$methode,$kritisch){
      $this->nummer=$nummer; $this->meldung=$meldung;
      $this->methode=$methode; $this->kritisch=$kritisch;
    }

    public function getNummer(){
      return $this->nummer;
    }
```

Listing 4.54: Die erste Fehlerklasse

```
    public function getMeldung(){
      return $this->meldung;
    }
    public function getMethode(){
      return $this->methode;
    }
    public function istKritisch(){
      return $this->kritisch;
    }
    public function __toString(){
      $ausgabe='Fehler Nr. '.$this->getNummer().'<br>';
      $ausgabe.='in Methode '.$this->getMethode().'<br>';
      $ausgabe.=$this->getMeldung().'<br>';
      return $ausgabe;
    }
  }
?>
```

Listing 4.54: Die erste Fehlerklasse (Forts.)

Nachdem die Fehlerklasse erstellt wurde, kann sie in Verbindung mit anderen, eigenen Klassen angewendet werden. Im ersten Beispiel wird die MySQL-Implementierung des Datenbank-Interfaces *iDZ* auf die objektorientierte Fehlerbehandlung umgestellt.

Die Methoden liefern nun im Fehlerfall nicht mehr *FALSE* zurück, sondern *NULL*. Wenn ein Fehler entsteht, weil beispielsweise das übergebene Parameterobjekt ungültig ist oder keine Verbindung zum Datenbankserver hergestellt werden kann, wird ein neues Fehlerobjekt erzeugt und geworfen. Der Befehl *throw new Fehler(...)* ruft den Konstruktor der Fehlerklasse auf. Im Anschluss daran wird die aufgerufene Methode wie bei einer *return*-Anweisung sofort beendet. Das Fehlerobjekt wird dabei automatisch an den Aufrufer weiter gegeben.

Im Gegensatz zu Listing 4.47 erkennen Sie, dass die Texte der neu erzeugten Fehlermeldungen (fett gedruckt) den Quelltext vergrößern. Bei der Fehlerbehandlung sind Sie jedoch auf aussagekräftige Texte angewiesen. Als Alternative könnten Sie auch nur Fehlernummern vergeben und jeder Nummer in einer externen Textdatei oder in einer MS-Excel-Liste einen Fehlertext zuweisen. Auf diese Weise kann mit mehreren externen Dateien auch eine Sprachumschaltung der Fehlermeldungen realisiert werden.

```
<?php
  class mysqlDZ implements iDZ{
    private $conn; private $connected=FALSE;

    public function öffnen($p){
      if (!isset($p)) throw new Fehler(1,'Das Parameter-Objekt ist
```

Listing 4.55: Datenbankzugriff mit objektorientiertem Fehlermanagement

```
                                   ungültig!','mysqlDZ-öffnen',FALSE);
    if ((!$p->isParam('host'))||(!$p->isParam('user'))||
                        (!$p->isParam('pass'))||(!$p->isParam('db'))){
      throw new Fehler(2,'Das Parameter-Objekt ist
                              unvollständig!','mysqlDZ-öffnen',FALSE);
    }
    $host=$p->get('host'); $user=$p->get('user');
    $pass=$p->get('pass'); $db=$p->get('db');
    $conn=@mysql_connect($host,$user,$pass);
    if ($conn){
      if (@mysql_select_db($db,$conn)!=1){
        throw new Fehler(3,'Datenbank '.$db.' konnte nicht geöffnet
                                     werden!','mysqlDZ-öffnen',FALSE);
      }
    }
    else{
      throw new Fehler(4,'Verbindung zum Server konnte nicht aufgebaut
                                     werden!','mysqlDZ-öffnen',FALSE);
    }
  }

  public function schliessen($p){
    @mysql_close();
  }

  public function lesen($p){
    if (!isset($p)) throw new Fehler(5,'Das Parameter-Objekt ist
                                    ungültig!','mysqlDZ-lesen',FALSE);
    if (!$p->isParam('sql')) throw new Fehler(6,'Das Parameter-Objekt
                            ist unvollständig!','mysqlDZ-lesen',FALSE);
    $sql=$p->get('sql');
    if ($this->startswith($sql,'SELECT')==FALSE){
      throw new Fehler(7,'Das SQL-Statement muss mit SELECT
                                    beginnen!','mysqlDZ-lesen',FALSE);
    }
    $data=@mysql_query($sql);
    if ($data==FALSE){
      throw new Fehler(8,'Ergebnis-Menge ist ungültig oder leer!
                                             ','mysqlDZ-lesen',FALSE);
    }
```

Listing 4.55: Datenbankzugriff mit objektorientiertem Fehlermanagement (Forts.)

```
      $ausgabe=Array();
      $x=0;
      while($row=mysql_fetch_row($data)){
        $datensatz=Array();
        for($i=0;$i<count($row);$i++){
          $datensatz[$i]=$row[$i];
        }
        $ausgabe[$x]=$datensatz;
        $x++;
      }
      return $ausgabe;
    }

    public function schreiben($p){
      if (!isset($p)) throw new Fehler(9,'Das Parameter-Objekt ist
                                 ungültig!','mysqlDZ-schreiben',FALSE);
      if (!$p->isParam('sql')) throw new Fehler(10,'Das Parameter-Objekt
                          ist unvollständig!','mysqlDZ-schreiben',FALSE);
      $sql=$p->get('sql');
      if (($this->startswith($sql,'UPDATE')==TRUE)||($this->startswith
                                              ($sql,'INSERT')==TRUE)){
        // UPDATE oder INSERT
        if (@mysql_query($sql)!=TRUE){
          throw new Fehler(11,'Fehler beim Ausführen der SQL-Anweisung!',
                                           'mysqlDZ-schreiben',FALSE);
        }
      }
      else{
        // FEHLER
        throw new Fehler(12,'Das SQL-Statement muss mit UPDATE oder
                           INSERT beginnen!','mysqlDZ-schreiben',FALSE);
      }
    }

    private function startswith($str,$wert){
      return strtolower(substr($str,0,strlen($wert)))==strtolower($wert);
    }
  }
?>
```

Listing 4.55: Datenbankzugriff mit objektorientiertem Fehlermanagement (Forts.)

Das Testskript für die Fehlerbehandlung aus Listing 4.48 muss nur unwesentlich verändert werden. Zunächst geht man davon aus, dass der Quellcode erfolgreich abgearbeitet wird. Die Abfragen, ob das Öffnen, Schreiben und Lesen erfolgreich war oder nicht, fallen demnach weg.

Im darauf folgenden Test wurde beim Öffnen der Verbindung das Parameterobjekt *$p_öffnen* nicht vollständig gefüllt; der Parameter für das Passwort wurde weggelassen. Das PHP-Skript beantwortet dies mit der folgenden Meldung:

*Fatal error: Uncaught Fehler Nr. 2
in Methode mysqlDZ-öffnen
Das Parameter-Objekt ist unvollständig!
 thrown in ... mysqlDZ.inc.php on line 9*

Es wurde also in Listing 4.55 der Fehler Nummer 2 erzeugt. Die *Öffnen*-Methode wurde abgebrochen. Dann wurde festgestellt, dass der Quellcode nicht in einem *try/catch*-Block ausgeführt wurde. Das entstandene Fehlerobjekt konnte dadurch nicht abgefangen werden. Dies führt zum Abbruch der Abarbeitung des Skripts und zur Ausgabe der Meldung.

Um einen Fehler abzufangen, versuchen Sie nun, den Quellcode erfolgreich auszuführen. Dazu wird er in eine *try*-Konstruktion geschrieben. Dieser Code wird in Listing 4.56 fett dargestellt. Sie erkennen, dass der Code nicht durch Fehlerabfragen unterbrochen wird.

```
<?php require_once("classloader.inc.php"); ?>
<html><body>
<?php
  $db=new mysqlDZ();
  try{
    // 1. öffnen
    $p_öffnen=new ParameterListe();
    $p_öffnen->add('host','localhost'); $p_öffnen->add('user','root');
    $p_öffnen->add('pass',''); $p_öffnen->add('db','boerse');
    $db->öffnen($p_öffnen);
    // 2. schreiben
    $p_schreiben=new ParameterListe();
    $p_schreiben->add('sql','UPDATE ag SET name="Dopatka AG" WHERE
                                                              ID=6');
    $db->schreiben($p_schreiben);
    // 3. lesen
    $p_lesen=new ParameterListe();
    $p_lesen->add('sql','SELECT ID,name FROM ag ORDER BY ID');
    $ausgabe=$db->lesen($p_lesen);
    echo 'Anzahl Datensätze:'.count($ausgabe).'<br>';
    echo 'Anzahl Spalten:'.count($ausgabe[0]).'<br>';
    foreach($ausgabe as $index => $datensatz){
```

Listing 4.56: Objektorientierte Fehlerbehandlung beim Datenbankzugriff

```
      foreach($datensatz as $index2 => $wert){
        echo $wert; echo '<br>';
      }
    }
    $db->schliessen(NULL);
  }
  catch(Fehler $f){
    echo $f;
    $db->schliessen(NULL);
  }
?>
</body></html>
```

Listing 4.56: Objektorientierte Fehlerbehandlung beim Datenbankzugriff

Wenn eine Anweisung wie *$db->öffnen($p_öffnen)* ein Fehlerobjekt zurückgibt, wird der reguläre Programmablauf sofort unterbrochen und der nächste passende *catch*-Block hinter dem *try*-Block aufgesucht. Passend bedeutet, dass die abgefangene Fehlerklasse (hier mit dem Namen *Fehler*) mit dem erzeugten Fehlerobjekt verglichen wird. Ist das erzeugte Fehlerobjekt ein Objekt der Klasse *Fehler*, so wird der *catch*-Block abgearbeitet. Im Beispiel von Listing 4.56 wird die *toString()*-Methode der Fehlerklasse ausgeführt und eine eventuell geöffnete Datenbankverbindung wieder geschlossen.

Wenn man in diesem Fall beispielsweise die Angabe des Parameters *pass* bei *$p_öffnen* vergisst, so wird unmittelbar nach dem Aufruf von *$db->öffnen($p_öffnen)* der *catch*-Block aufgerufen, der die folgende Ausgabe erzeugt:

Fehler Nr. 2

in Methode mysqlDZ-öffnen

Das Parameter-Objekt ist unvollständig!

Hinweis

Hinter einem *try*-Block können sich also mehrere *catch*-Blöcke befinden, die von oben nach unten durchsucht werden. Da Sie Fehlerklassen voneinander vererben können, wird nach der ersten passenden „Ist ein"-Beziehung gesucht. Ist eine solche Bedingung gefunden, wird der *catch*-Block abgearbeitet und hinter dem letzten *catch*-Block mit der Abarbeitung des Quellcodes fortgefahren. Es ist daher ratsam, im letzten *catch*-Block die Oberklasse *Exception* abzufangen und abzuarbeiten.

Auf eine ähnliche Weise kann auch die Flugreservierung aus Listing 4.51 auf eine objektorientierte Fehlerbehandlung umgestellt werden. Dabei kann dieselbe Fehlerklasse verwendet werden. In diesem Fall werden irreguläre Zustandsübergänge in Fehlerobjekte umgewandelt.

```
<?php
  class AirlineReservierung implements iFlugreservierung{
...
    public function reservieren(){
      if ($this->zustand!=0){
        throw new Fehler(20,'Reservieren im Zustand '.$this->getZustand()
                              .' nicht erlaubt!','Reservierung',FALSE);
      }
      $this->zustand=1; // reserviert
    }

    public function stornieren(){
      if ($this->zustand!=1){
        throw new Fehler(21,'Stornieren im Zustand '.
             $this->getZustand().' nicht erlaubt!','Reservierung',FALSE);
      }
      $this->zustand=3; // storniert
    }

    public function buchen(){
      if ($this->zustand!=1){
        throw new Fehler(22,'Buchen im Zustand '.$this->getZustand().
                             ' nicht erlaubt!','Reservierung',FALSE);
      }
      // es ist Zufall, ob die Buchung funktioniert...
      srand(microtime()*1000000);
      $zufall=rand(0,2);
      if ($zufall>1){
        $this->zustand=3; // gebucht
      }
      else{
        $this->zustand=2; // storniert
      }
    }
  }
?>
```

Listing 4.57: Implementierung der Flugreservierung mit objektorientiertem Fehlermanagement

Der Test der Implementierung in Listing 4.58 erfolgt wie gewöhnlich in einem *try/catch*-Block. In diesem Fall wird versucht, auf das Reservierungsobjekt die *reservieren*-Methode zweimal hintereinander aufzurufen.

```
<?php require_once("classloader.inc.php"); ?>
<html><body>
<?php
  try{
    $FDairline=new AirlineReservierung();
    echo $FDairline->getZustand().'<br>';
    $FDairline->reservieren();
    echo $FDairline->getZustand().'<br>';
    $FDairline->reservieren();
    echo $FDairline->getZustand().'<br>';
  }
  catch(Fehler $f){
    echo $f;
  }
?>
</body></html>
```

Listing 4.58: Test der Flugreservierung mit irregulären Methodenaufrufen

Beim Erzeugen des Reservierungsobjekts wechselt dies in den Zustand „initialisiert". Nach dem ersten, regulären Aufruf der Methode *reservieren* wechselt das Objekt in den Zustand *reserviert*. Wird das Reservieren dann nochmals aufgerufen, ist dies kein gültiger Pfad im Zustandsautomaten nach Abbildung 4.13. Daher wird der Aufruf mit einer Fehlermeldung quittiert:

initialisiert

reserviert

Fehler Nr. 20

in Methode Reservierung

Reservieren im Zustand reserviert nicht erlaubt!

Andere Fehler: php.ini und eigene Error Handler

Im Gegensatz zu anderen objektorientierten Programmiersprachen wie Java oder .NET ist das *try/catch*-Konzept von PHP 5 noch nicht vollständig implementiert. So bieten andere Sprachen zusätzlich einen *finally*-Block an für Befehle, die auf jeden Fall auszuführen sind und zwar unabhängig davon, ob ein Fehler aufgetreten ist oder nicht. Fehlt dieses Konzept in einer Sprache, so werden in einem Fehlerfall beispielsweise Datenbankverbindungen oft nicht korrekt geschlossen oder es bleiben temporäre Dateien bestehen. Dies sorgt wiederum für eine langfristige Instabilität der Anwendung. Hier müssen Sie als PHP-Entwickler besonders sorgfältig und gewissenhaft alle möglichen Fälle eines Programmablaufs beachten.

Eine weitere Schwachstelle im *try/catch*-Konzept von PHP 5 liegt darin, dass es nicht universell anwendbar ist. So führt unter anderem die Anweisung *$x=10/0;* zu der Ausgabe

Warning: Division by zero in ... .php on line 3, selbst wenn die Anwendung in einem *try*-Block steht. Das Konzept funktioniert in PHP also nur in Verbindung mit Objekten, die selbst Exceptions werfen.

Im selben Zusammenhang wirft beispielsweise Java selbstständig eine *NullPointerException*, die innerhalb des Anwendungskontexts gefangen werden kann. Bis zur derzeit aktuellen Version 5.3 von PHP müssen Sie hingegen auf das veraltete Reporting-Management von PHP zugreifen. Dabei unterscheidet PHP folgende Arten von Fehlern, deren Werte bereits als Konstanten hinterlegt sind:

Konstante	Wert	Bedeutung
E_ERROR	1	Fehler, die nicht behoben werden können; führt zum Abbruch des Skripts
E_WARNING	2	Laufzeitwarnungen, die nicht zum Abbruch des Skripts führen
E_PARSE	4	Parser-Fehler beim Interpretieren des Skripts; das Skript startet nicht
E_NOTICE	8	Benachrichtigungen während der Laufzeit, die nicht zum Abbruch des Skripts führen
E_CORE_ERROR	16	wie *E_ERROR*, wobei die Meldungen vom PHP-Kernel stammen
E_CORE_WARNING	32	wie *E_WARNING*, wobei die Meldungen vom PHP-Kernel stammen
E_COMPILE_ERROR	64	schwerer Fehler beim Übersetzen des Skripts, wobei die Meldungen der Zend Engine stammen, die für das Kompilieren des Skripts verantwortlich ist und gleichzeitig die Virtuelle Maschine (VM) von PHP darstellt
E_COMPILE_WARNING	128	Warnungen beim Übersetzen des Skripts, wobei die Meldungen von der Zend Engine stammen
E_USER_ERROR	256	eigene Fehlermeldungen, die Sie über den Befehl *trigger_error* erzeugen können
E_USER_WARNING	512	eigene Warnmeldungen, die Sie über den Befehl *trigger_error* erzeugen können
E_USER_NOTICE	1024	eigene Benachrichtigungen, die Sie über den Befehl *trigger_error* erzeugen können
E_STRICT	2048	enthält zusätzlich Vorschläge für bessere Kompatibilität des Codes
E_RECOVERABLE_ERROR	4096	potenziell gefährlicher Fehler aufgetreten, der die Engine aber nicht in einem instabilen Zustand hinterlassen hat; wird der Fehler nicht durch *set_error_handler()* behoben, so wird das Skript wie bei *E_ERROR* abgebrochen
E_DEPRECATED	8192	Warnung, dass der Quellcode in zukünftigen PHP-Versionen nicht mehr funktionieren wird

Tabelle 4.3: Übersicht über die Fehler- und Warnmeldungen von PHP

Konstante	Wert	Bedeutung
E_USER_DEPRECATED	16384	wie *E_DEPRECATED*, jedoch über den Befehl *trigger_error* selbst erzeugt; damit können Sie auf veralteten Quellcode im Kontext Ihrer eigenen Anwendung hinweisen
E_ALL	30719	alle Fehler und Warnungen, mit Ausnahme von *E_STRICT*

Tabelle 4.3: Übersicht über die Fehler- und Warnmeldungen von PHP (Forts.)

Welche Meldungen dargestellt werden, können Sie in der Konfigurationsdatei *php.ini* in dem Parameter *error_reporting* einstellen. Die Konfigurationsdatei befindet sich bei XAMPP im Unterverzeichnis *php*. Der betreffende, gut dokumentierte Eintrag sieht standardmäßig in der PHP-Version 5.30 folgendermaßen aus:

```
; Common Values:
;   E_ALL & ~E_NOTICE
;   (Show all errors, except for notices and coding standards warnings.)
;   E_ALL & ~E_NOTICE | E_STRICT  (Show all errors, except for notices)
;   E_COMPILE_ERROR|E_RECOVERABLE_ERROR|E_ERROR|E_CORE_ERROR
;   (Show only errors)
;   E_ALL | E_STRICT
;   (Show all errors, warnings and notices including coding standards.)
; Default Value: E_ALL & ~E_NOTICE
; Development Value: E_ALL | E_STRICT
; Production Value: E_ALL & ~E_DEPRECATED
; http://php.net/error-reporting
error_reporting = E_ALL & ~E_NOTICE & ~E_DEPRECATED
```

Da die Werte der Konstanten Zweierpotenzen darstellen, können mehrere Konstanten mit binären Operatoren hinzu- oder abgeschaltet werden. Abbildung 4.14 erklärt einige der binären Verknüpfungen und deren Ergebnis grafisch.

	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	2048	**1024**	**512**	**256**	**128**	**64**	**32**	**16**	**8**	**4**	**2**	**1**
E_ERROR	0	0	0	0	0	0	0	0	0	0	0	1
E_PARSE	0	0	0	0	0	0	0	0	0	0	1	0
UND (&)	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*1*	*1*
E_ALL	0	1	1	1	1	1	1	1	1	1	1	1
NOT E_NOTICE	1	1	1	1	1	1	1	1	0	1	1	1
UND (&)	*0*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*0*	*1*	*1*	*1*
E_ALL	0	1	1	1	1	1	1	1	1	1	1	1
E_STRICT	1	0	0	0	0	0	0	0	0	0	0	0
ODER (|)	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*

Abbildung 4.14: Binäre Verknüpfung von Fehlerkonstanten

Listing 4.59 zeigt einen Quellcode, bei dem das Ende der Zeichenkette in der *echo*-Anweisung nicht markiert ist. Dies führt bei der Standardeinstellung der *php.ini* zu dem Parser-Fehler *Parse error: syntax error, unexpected T_ENCAPSED_AND_WHITESPACE in .... on line 3.*

```
<html><body>
<?php
  echo 'Hallo?<br>;
?>
</body></html>
```

Listing 4.59: Eine fehlerhafte Anweisung

Sie können zwar versuchen, die Fehlerausgabe zu Beginn des Skripts mit der Anweisung *<?php ini_set("error_reporting",0); ?>* zu unterdrücken, da es sich jedoch um einen Parser-Fehler handelt, wird die Meldung trotzdem angezeigt, da das Skript ja gar nicht zur Ausführung kommt. Die Anweisung *ini_set* war ursprünglich dazu gedacht, für einzelne Skripte das Error Handling zu verändern, falls man selbst keinen Zugriff auf die *php.ini* besitzt, weil sie von einem Internetprovider verwaltet wird. Da die Anweisung jedoch selbst Teil des PHP-Skripts ist, ist der Effekt sehr begrenzt. In einer professionellen Anwendung sollten Sie also den Inhalt der *php.ini* selbst bestimmen können.

Die erste Idee, das Melden von Fehlern in der *php.ini* mit *error_reporting = 0* komplett abzuschalten, ist jedoch auch keine Lösung. In diesem Fall erzeugt das Skript nämlich gar keine Ausgabe, auch wenn andere PHP-Anweisungen vor der fehlerhaften Zeichenkette eine Anweisung hervorbringen würden. Das Behandeln des Fehlers ist hier also sehr umständlich. Der korrekte Ansatz besteht darin, Parser-Fehler durchaus auch in einer laufenden Anwendung zuzulassen, da sie auf einen Programmierfehler deuten, der zu beheben ist.

In der php.ini können Sie stattdessen die Bildschirmausgabe durch die Anweisung *display_errors = Off* abschalten. Um dennoch Fehler sehen zu können, sollten diese in eine Logdatei umgeleitet werden.

Dazu können Sie den Parameter *log_errors = On* der *ini*-Datei setzen. Nun müssen Sie noch die Ausgabe der Logdatei benennen. Dies geschieht durch den Parameter *error_log* und einer Dateiangabe, beispielsweise *error_log = "C:\php_error.log"*. Auf einem MS-Windows-Server können Sie das Logging durch *error_log = syslog* auch in die Ereignisanzeige des Betriebssystems umleiten.

Nun stellt sich noch die Frage, wie Sie einen benutzerdefinierten Fehler erzeugen können. Dies ist in Listing 4.60 dargestellt und funktioniert natürlich auch objektorientiert beim Aufruf von Methoden. Dort ist natürlich das Exception Handling der Vorgehensweise aus Listing 4.60 vorzuziehen. Das Skript liefert die Ausgabe *Fatal error: Kann nicht durch 0 teilen in ... on line 4.*

```
<html><body>
<?php
  function teile($x,$y){
    if ($y==0) trigger_error("Kann nicht durch 0 teilen", E_USER_ERROR);
    return($x/$y);
  }

  $a=4;
  $b=0;
  echo teile($a,$b);
?>
</body></html>
```

Listing 4.60: Ein selbsterzeugter Fehler

Sie können aber nicht nur Fehler selbst erzeugen, sondern die Verarbeitung von Fehlern in PHP auch umleiten. Für die Fehlergruppen

- *E_NOTICE*
- *E_WARNING*
- *E_USER_NOTICE*
- *E_USER_WARNING*
- *E_USER_ERROR*

können Sie eine Umleitung der Fehlerbehandlung in eine selbstdefinierte Funktion bewirken. Diese Funktion wird als Error Handler bezeichnet. Listing 4.62 zeigt die Definition eines solchen Handlers und das Umleiten auf diesen Handler mithilfe des PHP-Befehls *set_error_handler*.

```
<?php
  function meldung($typ, $meldung, $datei, $zeile){
   echo 'Typ: '.$typ.'<br>';
   echo 'Meldung: '.$meldung.'<br>';
   echo 'in Datei: '.$datei.'<br>';
   echo 'Zeile: '.$zeile.'<br>';
  }
  set_error_handler('meldung');
?>

<html><body>
  <?php echo $x; ?>
</body></html>
```

Listing 4.61: Eine eigene Fehlerbehandlung

Im HTML-Teil des Listings wird dann der Wert einer Variablen ausgegeben, die nicht zuvor deklariert wurde. Dies führt zu der folgenden Ausgabe:

Typ: 8

Meldung: Undefined variable: x

Datei: C:\Programme\xampp\htdocs\fehler\test1.php

Zeile: 12

Nun können Sie sich vorstellen, dass durch diesen Error Handler, den Sie in jedem Skript mit der *require*-Anweisung einbinden können, ein zentrales Fehlermanagement ermöglicht wird. So können Sie innerhalb dieses Handlers

- in eine Logdatei schreiben
- eine Datenbankverbindung eigens für das Fehlermanagement öffnen und den Fehler dort ablegen
- dem Administrator der Anwendung eine E-Mail senden

4.4 PHP und XML

Die Extensible Markup Language (XML) dient zur Beschreibung von baumförmig angeordneten Daten, die textbasiert dargestellt werden. Eine XML-Datei besteht also ausschließlich aus Textzeichen, die Sie mit einem gewöhnlichen Texteditor darstellen können.

Die Idee bei XML besteht darin, die Daten von ihrer Darstellung zu trennen. Ziel ist es, die Daten nach Wunsch als Liste, Tabelle oder auch als Grafik auszugeben und für alle Arten der Darstellung die gleiche Datenbasis im XML-Format zu nutzen.

Gerade bei großen Anwendungen wird auch häufig gefordert, einen Teil des Datenstamms zu im- oder exportieren. Hier hat sich das XML-Format als effektives, plattformunabhängiges Austauschprotokoll etabliert, das sich auch für den automatisierten Datenaustausch eignet.

Die nächste Frage, die sich stellt, ist die Verbindung von XML mit der objektorientierten Denkweise. XML stellt eine Baumstruktur dar, während Objekte sich gegenseitig beliebig kennen können. Die Antwort liegt darin, dass auch in einem XML-Baum Querreferenzen zwischen einzelnen Einträgen möglich sind. Sie können also eine Menge von Objekten, die sich gegenseitig kennen, in einem XML-Baum darstellen und umgekehrt. In Verbindung mit der Tatsache, dass Sie Objektgeflechte in XML speichern bzw. laden können und XML ein globales Austauschformat darstellt, können Sie XML als Transportmittel für Ihre Objektstrukturen von und zu anderen Systemen verwenden.

So ist es beispielsweise möglich, eine Liste von Kundenobjekten mit den Kundendaten wie Name, Vorname, Anschrift und Kundennummer in einer XML-Datei abzulegen. Zu jedem Kunden können im XML-Baum seine Rechnungen abgelegt werden, wobei jede Rechnung wieder aus Rechnungspositionen besteht, die jeweils einem Artikel zugeordnet sind (Abb. 4.10). Eine solche Struktur wird in Listing 4.62 skizziert.

```
<?xml version="1.0"?>
<kundenliste>
  <kunde id="1">
    <name>Dopatka</name>
    <vorname>Frank</vorname>
    <anschrift>...</anschrift>
    <rechnung id="1">
      <datum>05.02.2010</datum>
      <position id="1">
        <artikel id="32">
        <menge>3</menge>
      </position>
      <position id="2">
        <artikel id="53">
        <menge>1</menge>
      </position>
      ...
    </rechnung>
    <rechnung id="2">
      ...
    </rechnung>
  </kunde>
  <kunde id="2">
    ....
```

Listing 4.62: XML-Datei, die eine Objektinfrastruktur realisiert

```
  </kunde>
</kundenliste>
```

Listing 4.62: XML-Datei, die eine Objektinfrastruktur realisiert (Forts.)

Im Gegensatz zu HTML (Hypertext Markup Language) können Sie bei XML die Tags in den spitzen Klammern frei definieren. HTML ist also eine Untermenge von XML, bei der einzelnen Elementen eine bestimmte Bedeutung zugewiesen wurde. So bildet *<h1>* eine Überschrift und *<ul>* kennzeichnet den Beginn einer Aufzählung.

Sie können bei XML zusätzlich zu den Tags Attribute vergeben, wie es im Beispiel von Listing 4.62 mit den IDs der Kunden, Rechnungen, Positionen und Artikel geschehen ist. Zusätzlich können XML-Elemente Unterelemente enthalten, so beinhaltet eine Rechnung eine Menge von Rechnungspositionen. Als Alternative dazu kann zwischen einem Tags-Beginn und einem Tags-Ende ein Text stehen, wie im Beispiel *<name>Dopatka</name>*. Dies ist ebenso in HTML möglich, beispielsweise *<h1>Einleitung</h1>*.

wohlgeformt und gültig

Die Regeln für eine „gute" XML-Datei sind etwas strikter als beim gewohnten HTML-Format. So ist eine XML-Datei wohlgeformt, wenn sie sämtliche XML-Regeln einhält. Dies bedeutet, dass

- die Datei genau ein Wurzelelement besitzt. Das Wurzelelement ist das jeweils äußerste Element der Datei, im Beispiel des Listings 4.64 ist dies das Tag *<kundenliste>*.
- alle Elemente mit Inhalt eine Beginn- und eine Endkennung besitzen. So muss passend zum Tag *<name>* ein *</name>* existieren. Elemente ohne Inhalt können auch in sich geschlossen werden, indem die Endkennung vor der schließenden spitzen Klammer eingefügt wird. So ist der HTML-Zeilenumbruch *
* in XML nur dann gültig, wenn er *
* geschrieben wird.
- ein Tag nicht mehrere Attribute mit demselben Namen besitzen darf

Wenn eine bestehende XML-Datei eingelesen wird und deren Daten interpretiert werden, bezeichnet man dies als „Parsen". Ist eine XML-Datei nicht wohlgeformt, so bricht das einlesende Programm, der Parser, zumeist mit einer Fehlermeldung ab.

Es wurde bereits erwähnt, dass die Namen und Attribute der Elemente einer XML-Datei frei gewählt werden können. In der Realität schreibt eine Anwendung, die eine XML-Datei importiert, jedoch Regeln vor, die von der Datei eingehalten müssen.

Für diese Regeln existieren zwei verschiedene Sprachen, die beide weit verbreitet sind. Die DTD (Dokumenttypdefinition) soll langfristig von den XML Schemata abgelöst werden, die eine wesentlich genauere Definition der Inhalte einer XML-Datei zulassen. Ein weiterer Vorteil eines XML Schemas besteht darin, dass es selbst im XML-Format verfasst ist. Die entsprechenden Elemente sind jedoch vordefiniert.

Sie können nun eine XML-Datei gegen solch ein Regelwerk automatisch prüfen lassen. Entweder entspricht diese Datei den Regeln, so ist diese XML-Datei gemäß dem entsprechenden Regelwerk gültig, ansonsten nicht. Die Regelwerke werden meist als Dokumen-

tation zur Verfügung gestellt, um anderen Anwendungen – die natürlich auch in einer anderen Programmiersprache verfasst sein können – eine Anleitung für einen erfolgreichen Import-Vorgang zu liefern.

Grundlagen des Parsens: Ströme vs. Bäume

Grundsätzlich existieren zwei verschiedene Verfahren, um XML-Dateien zu verarbeiten, also zu parsen. Die erste Art besteht darin, eine XML-Datei als Datenstrom aufzufassen. Diese Idee wird von SAX-Parsern (Simple API for XML) verfolgt. Bei SAX handelt es sich um einen De-facto-Standard, der sich an den Elementen der XML-Datei orientiert.

Die Datei wird dabei automatisch eingelesen. Trifft der SAX-Parser auf ein *Beginn*- oder ein *Ende*-Element, so wird jeweils eine bestimmte Methode aufgerufen, die Sie implementieren müssen. Dies gilt ebenso, wenn Daten zwischen einem Element erkannt wurden, wie in *<name>Dopatka</name>*. In Abhängigkeit von dem Zustand des Einlesens, den Sie selbst definieren und verwalten müssen, sind die Daten der Elemente dann auszuwerten. Abbildung 4.15 skizziert, wie der Beginn des Eintrags *<vorkenntnisse>* vom SAX-Parser erkannt wird und eine Weiterleitung an die selbst zu implementierende PHP-Methode *start* erfolgt.

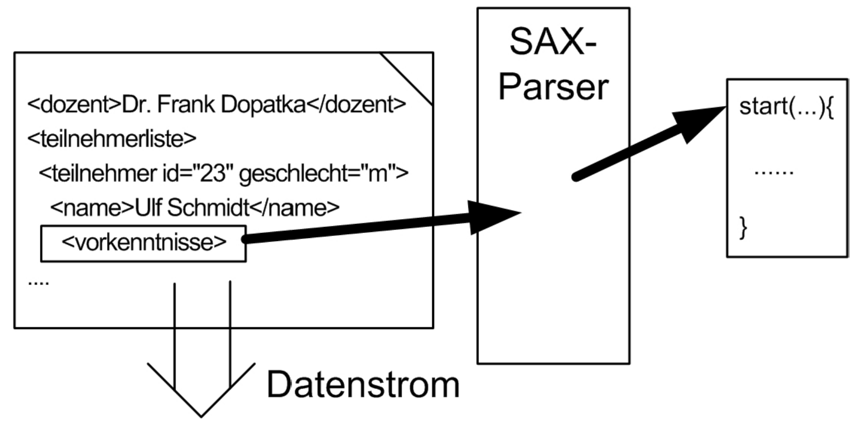


Abbildung 4.15: Prinzip eines SAX-Parsers

Prinzipiell benötigen SAX-Parser wenig Speicher und sind schnell in der Verarbeitung der Daten. Andererseits ist die Interpretation der Daten oft aufwendig zu implementieren und Sie besitzen keine Sicht auf den gesamten XML-Baum. SAX-Parser eignen sich besonders dann, wenn Sie in einer XML-Datei nach Inhalten suchen oder nur einen Teil der XML-Datei auslesen wollen.

Im Gegensatz zu SAX wird das DOM (Document Object Model) vom W3C (World Wide Web Consortium) als Organisation definiert. Das Ziel eines DOM-Parsers besteht darin, die Baumstruktur der XML-Datei im Speicher nachzubilden, wobei jeder Knoten und jedes Blatt aus einem Objekt besteht. Die wichtigsten Knoten im DOM sind

- der Dokumentknoten, der die gesamte Baumstruktur darstellt
- Dokumentfragmentknoten, die jeweils einen Teil der Baumstruktur darstellen

- Elementknoten, die jeweils exakt einem XML-Element entsprechen
- Attributknoten, die jeweils exakt einem Attribut in XML entsprechen
- Textknoten, welche den textuellen Inhalt eines Elements oder Attributs darstellen

Ein DOM-Parser liest im ersten Schritt die XML-Datei ein und erzeugt ein Dokumentenobjekt. Über dieses Objekt können Sie nun mittels der DOM-Methoden auf die Inhalte und auf die Struktur des Datenbaums zugreifen. Dazu gehören vor allem

- die Navigation zwischen den einzelnen Knoten des Baums
- das Erzeugen, Verschieben und Löschen von Knoten
- das Lesen, Ändern und Löschen von Texten

Bei der Verwendung von DOM steht Ihnen also der gesamte Inhalt der XML-Datei jederzeit zur Verfügung, da sich die Struktur vollständig im Arbeitsspeicher befindet. Dies ist jedoch gleichzeitig auch der Nachteil von DOM. Da XML-Dateien sehr groß werden können und die Erzeugung der DOM-Objekte im Speicher zusätzlichen Overhead belegen, verwendet DOM wesentlich mehr Ressourcen als SAX. Dafür erhält man andererseits die volle Kontrolle über die Datenstruktur.

Abbildung 4.16 zeigt, wie eine XML-Datei von einem DOM-Parser eingelesen wird. Dabei wird der XML-Baum im Arbeitsspeicher des Servers aufgebaut, auf den man dann mit vorgegebenen Methoden zugreifen kann.

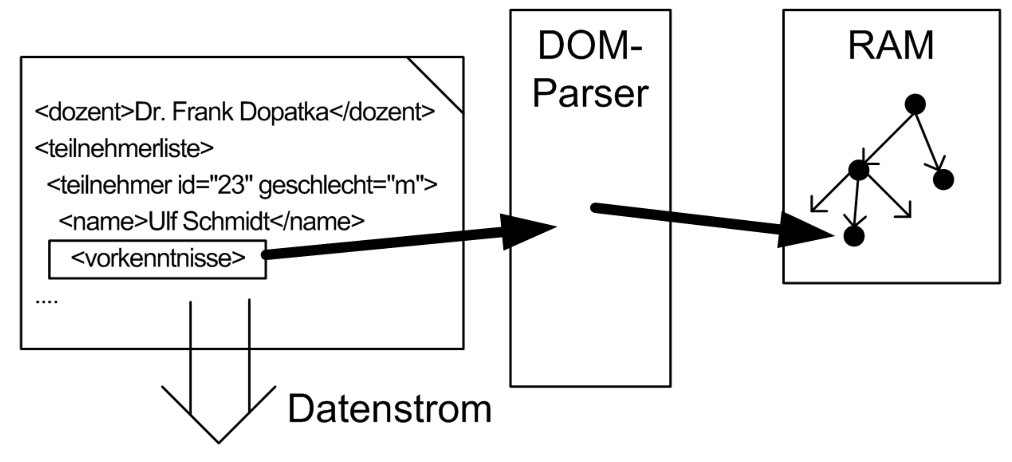


Abbildung 4.16: Prinzip eines DOM-Parsers

Profitipp

Sie müssen anhand Ihres Anwendungsfalls entscheiden, ob ein SAX- oder DOM-Parser für Ihre Anwendung besser geeignet ist. Dabei müssen Sie die Vor- und Nachteile der beiden Parser-Technologien abwägen und dabei die mögliche Größe der XML-Dokumente im Auge behalten. Eine nachträgliche Änderung des Parser-Standards bedeutet in der Regel einen großen Änderungsaufwand in Ihrer Anwendung!

Die beispielhafte XML-Datei in Listing 4.63 wird im Folgenden verwendet, um die Implementierung des SAX- und DOM-Parsers in PHP 5 zu testen. Die Unterstützung des XML-Formats wurde in PHP 5 weitreichend verbessert.

Bei der Datei handelt es sich um einen Ausschnitt aus einer Seminarverwaltung, bei der neben dem Namen des Seminars auch der Dozent und die optionale Teilnehmerliste verwaltet werden. Jeder Teilnehmer kann bei seiner Anmeldung Vorkenntnisse angeben, die ebenfalls in der XML-Datei gespeichert werden.

```
<?xml version="1.0"?>
<seminare>
  <seminar id="S1120">
    <name>PHP5 objektorientiert</name>
    <dozent>Dr. Frank Dopatka</dozent>
    <teilnehmerliste>
      <teilnehmer id="23" geschlecht="m">
        <name>Ulf Schmidt</name>
        <vorkenntnisse>
          <vorkenntnis>Grundlagen HTML</vorkenntnis>
          <vorkenntnis>Java</vorkenntnis>
        </vorkenntnisse>
      </teilnehmer>
      <teilnehmer id="43" geschlecht="w">
        <name>Clair Grube</name>
        <vorkenntnisse>
          <vorkenntnis>JavaScript</vorkenntnis>
        </vorkenntnisse>
      </teilnehmer>
      <teilnehmer id="56" geschlecht="w">
        <name>Ulla Hansen</name>
      </teilnehmer>
      <teilnehmer id="53" geschlecht="m">
        <name>Franz Streber</name>
        <vorkenntnisse>
          <vorkenntnis>Visual Basic 6</vorkenntnis>
        </vorkenntnisse>
      </teilnehmer>
      <teilnehmer id="98" geschlecht="m">
        <name>Hans Wurst</name>
      </teilnehmer>
    </teilnehmerliste>
```

Listing 4.63: Beispielhafte XML-Datei

```
  </seminar>
</seminare>
```

Listing 4.63: Beispielhafte XML-Datei (Forts.)

Profitipp

Es kann sein, dass die Verarbeitung von großen XML-Dateien einige Zeit in Anspruch nimmt. Dann kann es vorkommen, dass Sie eine Timeout-Meldung in ihrem Web-Browser erhalten. Die Ursache liegt darin, dass in der *php.ini* eine maximale Ausführungszeit für ein PHP-Skript von 30 oder 60 Sekunden im Parameter *max_execution_time* vorgesehen ist. Dadurch wird eine hohe Serverlast durch fehlerhafte PHP-Skripte verhindert. Diesen Wert können Sie bei Bedarf erhöhen, beispielsweise auf 600, um ein PHP-Skript maximal 10 Minuten auszuführen. Ein weiterer Timeout liegt beim HTTP-Server. So ist bei dem Apache-Web-Server in der *httpd.conf* der Parameter Timeout standardmäßig auf 300 Sekunden, also auf 5 Minuten, für eine laufzeitbedingte Unterbrechung gesetzt. Auch hier können Sie bei Bedarf eine Erhöhung vornehmen.

XML als Datenstrom: SAX

Im ersten Beispiel wird die Funktionsweise des SAX-Parsers von PHP 5 erläutert. Dazu wird zunächst eine Teilnehmerklasse benötigt, die für das Beispiel stark vereinfacht wurde. Um auf die zahlreichen Get- und Set-Methoden zu verzichten, wurden die Eigenschaften der Klasse in Listing 4.64 – entgegen dem Prinzip der Datenkapselung der Objektorientierung – als *public* deklariert. Ein Teilnehmer besteht dabei aus

- einem Namen
- einem Geschlecht
- einer optionalen Liste seiner Vorkenntnisse

Die definierte *toString()*-Methode gibt alle Daten des erzeugten Teilnehmerobjekts aus.

```
class Teilnehmer{
  public $name=null; public $geschlecht=null; public $vorkenntnisse="";

  function __toString(){
    $ausgabe='Name: '.$this->name.'<br>';
    $ausgabe.='Geschlecht: '.$this->geschlecht.'<br>';
    if ($this->vorkenntnisse==""){
      $ausgabe.='Keine Vorkenntnisse!<br><br>';
    }
    else{
      $ausgabe.='Vorkenntnisse:<br>'.$this->vorkenntnisse.'<br>';
```

Listing 4.64: Die einfache Klasse „Teilnehmer“

```
    }
    return $ausgabe;
  }
}
```

Listing 4.64: Die einfache Klasse „Teilnehmer" (Forts.)

Das Ziel besteht darin, aus der XML-Datei aus Listing 4.63 Teilnehmerobjekte zu erzeugen, die dann weiterverwendet werden können. Dazu wird noch eine zweite Klasse benötigt, deren Objekt später dem XML-Parser zugewiesen wird. Die Parser-Klasse aus Listing 4.65 besteht aus drei Methoden. Die Methode *start(...)* wird aufgerufen, wenn der Parser ein öffnendes XML-Element wie *<teilnehmer>* identifiziert, die Methode *ende(...)* bei einem schließenden Element wie *</name>* und die Methode *cData*, wenn Textdaten in einem XML-Element gefunden werden. Beispielsweise bei *<name>Dopatka</name>* sind die Textdaten „Dopatka".

```
class ParserKlasse{
  private $aktuell=null; private $daten=null;
  private $tn=array();

  function start($p,$name,$atts){
    $this->daten=null;
    switch ($name){
      case 'TEILNEHMER':
        $this->aktuell=$atts['ID'];
        $this->tn[$this->aktuell]=new Teilnehmer();
        $this->tn[$this->aktuell]->geschlecht=$atts['GESCHLECHT'];;
        break;
    }
  }

  function ende($p,$name){
    if ($this->aktuell == null) return true;
    switch ($name){
      case 'NAME':
        $this->tn[$this->aktuell]->name=$this->daten;
        break;
      case 'VORKENNTNIS':
        $this->tn[$this->aktuell]->vorkenntnisse.=$this->daten.'<br>';
        break;
      case 'VORKENNTNISSE':
        $this->aktuell=null;
        break;
```

Listing 4.65: Die Hilfsklasse für den SAX-Parser

```
    }
  }

  function cData($p,$data){
    $daten=trim($data);
    if (!empty($daten)) $this->daten=$daten;
  }

  function getTN(){
    return $this->tn;
  }
}
```

Listing 4.65: Die Hilfsklasse für den SAX-Parser (Forts.)

Wie funktioniert nun diese Hilfsklasse? Als öffnendes XML-Element wird nur *<teilnehmer>* betrachtet, alle anderen öffnenden Elemente werden ignoriert. Den Namen des Elements und dessen Attribute gelangen als Input-Parameter in die Methode. Der SAX-Parser befüllt die Parameter automatisch, sodass Sie sich darum nicht kümmern müssen. Als Attribute des Elements *Teilnehmer* werden die Teilnehmer-ID und das Geschlecht des Teilnehmers festgehalten. Diese Attribute werden aus der XML-Datei extrahiert.

Aus der ID wird der Index des Datenfelds *$tn* bestimmt, damit der Teilnehmer eindeutig wiedergefunden werden kann. Im Anschluss daran wird ein neues Teilnehmerobjekt angelegt und dem Feld zugewiesen. Abschließend wird das Geschlecht des existierenden Teilnehmers mit dem gleichnamigen Attribut aus der XML-Datei belegt, das ebenso wie die ID über das Feld *$atts* aus der XML-Datei ausgelesen wurde.

Wenn der Parser Text zwischen einem öffnenden und einem schließenden Element ermittelt hat, entfernt die Methode *cData(...)* führende und folgende Leerzeichen durch die *trim*-Methode. Die Daten werden dann temporär in der Eigenschaft *$daten* abgelegt.

Die Methode *ende* wertet die schließenden Elemente *</name>*, *</vorkenntnis>* und *</vorkenntnisse>* aus. Wird das schließende Element *</name>* entdeckt, wird die Zeichenkette aus *cData(...)*, die sich in der Eigenschaft *$daten* befindet, dem aktuellen Teilnehmer zugewiesen. Beim schließenden Element *</vorkenntnis>* wird die ermittelte Vorkenntnis aus *$daten* der Liste der Vorkenntnisse des Teilnehmers hinzugefügt. Diese Liste besteht zur Vereinfachung lediglich aus einer Zeichenkette, wobei die Vorkenntnisse durch einen HTML-Zeilenumbruch getrennt sind. In einer realen Anwendung würde dafür natürlich ein separates Datenfeld verwendet. Ist die Liste der Vorkenntnisse in der XML-Datei durch *</vorkenntnisse>* vollständig, so ist der aktuelle Teilnehmer vollständig abgearbeitet.

Nach dem Parsen steht die Liste der Teilnehmer zur Verfügung, die über die Get-Methode *getTN()* abgerufen werden können.

Listing 4.66 testet den SAX-Parser von PHP. Dazu werden zunächst ein Parser erzeugt und die Methoden definiert, die auf die öffnenden und schließenden Elemente sowie auf Daten innerhalb der Elemente reagieren sollen.

Im Anschluss daran wird ein Objekt der Hilfsklasse aus Listing 4.65 erzeugt und dem Parser zugewiesen.

Nun kann die XML-Datei zum Lesen geöffnet werden. Es werden jeweils 1 024 Byte ausgelesen und dem Parser über den Befehl *xml_parse* zugeführt. Der Aufruf der Methoden der Hilfsklasse erfolgt automatisch über den Parser. Sobald der gesamte Dateiinhalt durch den SAX-Parser gelaufen ist, kann die Datei wieder geschlossen werden.

Im HTML-Teil des Testprogramms stehen jetzt die ausgelesenen Teilnehmerdaten zur Verfügung. Das Datenfeld *$tn* der Teilnehmer wird über das Hilfsobjekt abgeholt und die Daten der Teilnehmer werden in der *foreach*-Schleife ausgegeben.

```
<?php require_once("classloader.inc.php"); ?>
  $parser=xml_parser_create();
  xml_set_element_handler($parser,'start','ende');
  xml_set_character_data_handler($parser,'cData');
  $parserObjekt=new ParserKlasse();
  xml_set_object($parser,$parserObjekt);

  $fp=fopen('beispiel.xml','r');
  while($data=fread($fp,1024)){
    $result=xml_parse($parser,$data);
    if ($result==FALSE){
      die(sprintf("XML FEHLER: %s in Zeile %d",
      xml_error_string(xml_get_error_code($parser)),
      xml_get_current_line_number($parser)));
    }
  }
  fclose($fp);
?>

<html><body>
<?php
  $tn=$parserObjekt->getTN();
  foreach ($tn as $index => $wert){
    echo 'Teilnehmer ID '.$index.'<br>'; echo $wert;
  }
?>
</body></html>
```

Listing 4.66: Test des SAX-Parsens

Die Ausgabe besteht aus der Teilnehmer-ID, gefolgt von dem Aufruf der *toString()*-Methode jedes Teilnehmers:

Teilnehmer ID 23

Name: Ulf Schmidt
Geschlecht: m
Vorkenntnisse:
Grundlagen HTML
Java
Teilnehmer ID 43
Name: Clair Grube
Geschlecht: w
Vorkenntnisse:
JavaScript
...

XML als Baum: DOM

Die zweite Art der XML-Verarbeitung, die PHP 5 bietet, ist die Rekonstruktion des XML-Baums aus den Daten der XML-Datei im Arbeitsspeicher des Servers.

Als Beispiel wird wiederum die XML-Datei aus Listing 4.63 verwendet. Sobald die Datei geladen wurde, stehen deren Inhalte in der Objektreferenz *$doc* zur Verfügung. Wie Sie erkennen, ist die *load*-Methode eine statische Methode der Klasse *DOMDocument* und *$doc* ein Objekt dieser Klasse, das über eine Vielzahl von Methoden verfügt, um auf die Daten zuzugreifen.

Eine dieser Methoden lautet *getElementsByTagname*. Dabei werden alle Inhalte eines Elementtyps in einer Liste zurückgegeben. Im Beispiel in Listing 4.67 werden alle Vorkenntnisse der Teilnehmer eines Seminars in der Liste *$vorkenntnisse* gespeichert und im HTML-Teil des Skripts nacheinander mit Zeilenumbruch ausgegeben.

```
<?php
  $doc=DOMDocument::load('beispiel.xml');
  $vorkenntnisse=$doc->getElementsByTagname('vorkenntnis');
?>

<html><body>
<?php
  echo 'Anzahl der Vorkenntnisse: '.$vorkenntnisse->length.'<br>';
  for($i=0;$i<$vorkenntnisse->length;$i++){
    $kenntnis=$vorkenntnisse->item($i);
    echo $kenntnis->textContent.'<br>';
  }
?>
</body></html>
```

Listing 4.67: Ein einfaches DOM-Parsen

Aus den Daten der eingelesenen XML-Datei ergibt sich dann die folgende Ausgabe:

Anzahl der Vorkenntnisse: 4

Grundlagen HTML

Java

JavaScript

Visual Basic 6

Die einzelnen Methoden von *DOMDocument* hier aufzulisten und zu beschreiben, würde einige Seiten füllen. Kein Entwickler lernt diese Methoden auswendig, sondern sucht bei Bedarf die Methoden aus einer Onlinedokumentation, die ihm bei der Lösung seiner konkreten Problemstellung behilflich sind. Die bereitgestellten Dienste von *DOMDocument* können Sie beispielsweise unter *http://de3.php.net/book.dom* nachlesen. Bitte achten Sie dabei auf die Endung *dom*, und nicht *com* der Homepage! Dort befinden sich auch zahlreiche Quellcodebeispiele zum Umgang mit DOM.

Profitipp

In einer prozeduralen Programmiersprache kann ein Entwickler nach einigen Jahren Erfahrung meist die gesamte Sprache auswendig. In der Objektorientierung ist dies aufgrund der Vielzahl der Klassen und Methoden für verschiedene Zwecke nicht mehr möglich. Hier gilt die Regel: Sie müssen nur wissen, wo die Funktionen stehen, die Sie benötigen! Und Sie müssen so viel von der Sprache PHP beherrschen, dass Sie die Definitionen der Funktionalität verstehen und anwenden können!

In dem nächsten Beispiel wird die DOM-Klasse dazu verwendet, zunächst ein DOM-Objekt im Speicher des Servers aufzubauen und dieses Objekt anschließend als XML-Datei abzuspeichern.

Auf diese Weise können Sie genausogut alle Eigenschaften eines Objekts über eine selbstdefinierte Methode in ein DOM-Objekt überführen, um die Persistenz des Objekts sicherzustellen.

In Listing 4.68 erzeugen Sie zunächst ein neues DOM-Dokument in der XML-Version 1.0 mit westeuropäischem und amerikanischem Zeichensatz. Das XML-Dokument soll menschenlesbar formatiert werden, was sich insbesondere in entsprechenden Zeilenumbrüchen hinter den *Ende*-Elementen und in Einrückungen bei verschachtelten Elementen bemerkbar macht. Wenn Sie die XML-Datei ausschließlich maschinell verarbeiten, können Sie auf die Formatierung verzichten, die Verarbeitung etwas beschleunigen und die Dateigröße leicht minimieren.

Jedes XML-Element wird über die DOM-Methode *createElement* erzeugt und mit der Methode *appendChild* dem DOM-Baum hinzugefügt. Die Methode *setAttribute* definiert den Namen und den aktuellen Wert eines Attributs in einem XML-Element. Wenn Sie *createTextNode* verwenden, können Sie Text hinter dem zuletzt geöffneten XML-Element platzieren. Dies ist beispielsweise bei dem Namen des Seminars der Fall.

Wenn Sie mehrere Elemente befüllen, können Sie die Methoden *createElement* und *appendChild* in einer Schleife anwenden, um beispielsweise zuvor aus einer Datenbank ausgelesene Werte einem XML-Dokument hinzuzufügen.

Die Methode *saveXML* speichert das DOM-Objekt nicht in einer Datei, sondern erzeugt einen Datenstrom. Im HTML-Teil von Listing 4.68 wird dieser Datenstrom in eine neue Datei umgeleitet, sodass letztlich eine XML-Datei im Dateisystem des Webservers entsteht.

```
<?php
  $dom=new DOMDocument('1.0','iso-8859-1');
  $dom->formatOutput=TRUE;
  // Wurzel erzeugen:
  $seminare=$dom->createElement('seminare');
  $dom->appendChild($seminare);
  // Seminar erzeugen:
  $php=$dom->createElement('seminar');
  $php->setAttribute('id','S1120');
  // Name des Seminars und des Dozenten festlegen:
  $name=$dom->createElement('name');
  $name->appendChild($dom->createTextNode('PHP5 objektorientiert'));
  $php->appendChild($name);
  $doz=$dom->createElement('dozent');
  $doz->appendChild($dom->createTextNode('Dr. Frank Dopatka'));
  $php->appendChild($doz);
  // Teilnehmer-Liste anlegen:
  $tn_liste=$dom->createElement('teilnehmerliste');
  // Teilnehmer-Daten befüllen
  $php->appendChild($tn_liste);
  // XML-Dokument zusammensetzen
  $seminare->appendChild($php);
  $dom->appendChild($seminare);
  $daten=$dom->saveXML();
?>
<html><body>
<?php
  $fh=fopen('neu.xml','wb');
  if (fwrite($fh,$daten)===false){
    echo 'Konnte XML-Datei nicht schreiben!';
  }
  else{
    echo 'XML-Datei erfolgreich geschrieben!';
```

Listing 4.68: Mit DOM eine XML-Datei schreiben

```
  }
  fclose($fh);
?>
</body></html>
```

Listing 4.68: Mit DOM eine XML-Datei schreiben (Forts.)

Die geschriebene XML-Datei ist ein vereinfachtes Beispiel der Seminardatei und wird in Listing 4.69 dargestellt.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<seminare>
  <seminar id="S1120">
    <name>PHP5 objektorientiert</name>
    <dozent>Dr. Frank Dopatka</dozent>
    <teilnehmerliste/>
  </seminar>
</seminare>
```

Listing 4.69: Die geschriebene XML-Datei

Geprüftes XML: DTD und Schema

Wie Sie in den letzten Beispielen bereits gesehen haben, können Sie die Namen der Elemente frei vergeben. In diesem Kapitel werden Sie nun sehen, wie Sie anhand einer Dokumenttypdefinition oder anhand eines XML Schemas Regeln für diese Elemente vergeben können. Diese Regeln werden meist in einer separaten Datei mit der Endung *.dtd* bzw. *.xsd* abgelegt.

PHP bietet Ihnen die Möglichkeit, durch das Parsen nicht nur die Wohlgeformtheit der XML-Datei sicherzustellen, sondern auch die Gültigkeit in Bezug auf einen solchen Satz von Regeln.

Um die Beispieldatei nach einer DTD mit PHP 5 prüfen zu lassen, müssen Sie zunächst die Dokumenttypdefinition angeben, nach der Sie die Prüfung vornehmen wollen. Listing 4.70 zeigt die Verbindung der *beispiel.xml* mit der *seminare.dtd* Datei. Mit *seminare* ist der notwendige Wurzelknoten, also das oberste Element der XML-Datei, gemeint. An dieser Stelle beginnt auch die Beschreibung der Regeln innerhalb der DTD.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE seminare SYSTEM "seminare.dtd">
<seminare>
  <seminar id="S1120">
  ...
```

Listing 4.70: Modifikation der XML-Datei für eine DTD-Prüfung

Als Nächstes müssen in Listing 4.71 die Regeln definiert werden. Die DTD ist eine eigene Sprache mit eigener Syntax. Das Wurzelelement *seminare* besteht aus Elementen *seminar,*

wobei mindestens ein Seminar angegeben werden muss. Dies ist an dem + im Listing zu erkennen.

Ein Seminar besteht wiederum aus genau einem Namen, genau einem Dozenten und optional (?) aus einer Teilnehmerliste. Außerdem hat ein Seminar zwangsweise eine ID.

Bei dem Namen und dem Dozenten handelt es sich um beliebigen Text und/oder Zahlen. Das Format *PCDATA* (parsed character data) kann leider nicht genauer spezifiziert werden, was beim Einsatz eines Schemas möglich wäre.

Die Teilnehmerliste kann leer sein oder aus beliebig vielen Teilnehmern (*) bestehen. Beim Teilnehmer muss wiederum ein Name angegeben werden und keine bzw. genau eine Liste von Vorkenntnissen. Der Teilnehmer besitzt zwei weitere Attribute, die zwingend angegeben werden müssen, nämlich eine ID und das Geschlecht. Über das Schlüsselwort *#IMPLIED* können Sie übrigens optionale Attribute definieren.

Die Liste der Vorkenntnisse besteht aus Elementen vom Typ *Vorkenntnis*. Dieses Element besteht wiederum nur aus Text, der mit Zahlen vermischt sein kann.

```
<!ELEMENT seminare (seminar+)>
<!ELEMENT seminar (name,dozent,teilnehmerliste?)>
<!ATTLIST seminar
  id NMTOKEN #REQUIRED
>
<!ELEMENT name (#PCDATA)>
<!ELEMENT dozent (#PCDATA)>
<!ELEMENT teilnehmerliste (teilnehmer*)>
<!ELEMENT teilnehmer (name,vorkenntnisse?)>
<!ATTLIST teilnehmer
  id NMTOKEN #REQUIRED
  geschlecht NMTOKEN #REQUIRED
>
<!ELEMENT vorkenntnisse (vorkenntnis+)>
<!ELEMENT vorkenntnis (#PCDATA)>
```

Listing 4.71: Die passende DTD

Listing 4.72 zeigt, wie das Einlesen der XML-Datei mit anschließender Prüfung vollzogen wird. Wie gewöhnlich wird das XML-Dokument mit dem DOM-Parser aus dem Dateisystem des Servers in den Speicher geladen. Dabei erfolgt das Prüfen auf Wohlgeformtheit. Der Dokumentbaum kann nun über die Methode *validate()* des DOM-Objekts sehr leicht auf Gültigkeit geprüft werden, Sie müssen also keine weitere Programmierung vornehmen.

Dabei wird gegen die DTD geprüft, die in der XML-Datei angegeben wurde. Die *validate*-Methode liefert *TRUE* zurück bei einer erfolgreichen Prüfung, ansonsten *FALSE*.

```
<?php
  $doc=DOMDocument::load('beispiel.xml');
?>

<html><body>
<?php
  if ($doc->validate()){
    echo 'Die Datei beispiel.xml ist gültig.';
  }
  else{
    echo 'Die Datei beispiel.xml NICHT ist gültig!';
  }
?>
</body></html>
```

Listing 4.72: Prüfen der XML-Datei anhand der DTD auf Gültigkeit

Wenn Sie anstelle der DTD ein XML Schema zur Prüfung Ihrer XML-Dateien verwenden möchten, können Sie wesentlich präzisere Prüfungen vornehmen. Sie können beispielsweise Formate für Telefonnummern, Datums- und Währungsangaben oder für E-Mail-Adressen vorgeben. Außerdem ist ein XML-Schema selbst eine XML-Datei mit einem vordefinierten Satz von Elementen, ähnlich wie bei einer HTML-Datei das *<h1>* eine vordefinierte Bedeutung hat, nämlich die einer Hauptkapitelüberschrift.

Andererseits ist aufgrund der vielen Möglichkeiten die Sprache des XML Schemas auch deutlich komplexer und schwieriger von einem Menschen zu lesen und zu schreiben.

Listing 4.73 zeigt das entsprechende XML Schema für die Seminardatei, die noch nicht einmal wesentlich präziser ist als die DTD aus Listing 4.71. Erkennen Sie die komplexere Syntax?

Zunächst werden die XML-Elemente *Dozent*, *Name*, *Vorkenntnis* und *Seminar* vergeben. Ein Seminar besteht aus einem Namen, einem Dozenten und einer Teilnehmerliste, die später definiert wird. Ein Element wie das Seminar, das aus anderen Elementen besteht, wird als „komplexer Typ“ bezeichnet. Dabei kann mit *mixed="true"* noch angegeben werden, dass die in dem komplexen Typ enthaltenen Elemente in einer beliebigen Reihenfolge angeordnet werden können.

Zusätzlich besitzt ein Seminar ein Attribut *ID*, das stets angegeben werden muss (*use="required"*). Die Alternative dazu ist *use="optional"*. Mit *type="xs:string"* wird der Datentyp des Attributs festgelegt. XML-Schema besitzt ähnlich wie in einer Programmiersprache die vordefinierten Datentypen

- *xs:string*
- *xs:decimal*
- *xs:integer*
- *xs:float*

- *xs:boolean*
- *xs:date*
- *xs:time*

Sie können sich jedoch auch weitere Datentypen definieren. Im Fall des Seminars wurde lediglich *xs:string* verwendet, obwohl ein Seminar stets aus dem Buchstaben *S*, gefolgt von einer Zahl, besteht. Die Angabe im XML Schema ist zwar korrekt, könnte aber noch präzisiert werden, da beispielsweise andere Buchstaben als „S“ oder weitere Buchstaben nicht in der ID gestattet sind.

Im nächsten Schritt definieren Sie, dass Seminare aus einer Liste von Elementen des Typs *Seminar* bestehen.

Nun wird beschrieben, wie ein Teilnehmer auszusehen hat. Er besteht aus einem Namen und Vorkenntnissen, wobei Vorkenntnisse nicht zwingend erforderlich sind (*minOccurs= "0"*). Sie können also neben Attributen auch Elemente als *optional* deklarieren. Zusätzlich besteht ein Teilnehmer noch aus den zwei Pflichtattributen *Geschlecht* und *ID*, die ebenfalls noch genauer spezifiziert werden könnten.

Bei dem Geschlecht ist diese Spezifizierung exemplarisch vorgenommen worden. Dazu definieren Sie einen neuen Datentyp, der *gesch* genannt wurde. Er basiert auf einer Zeichenkette (*base="xs:string"*), deren Werte jedoch eingeschränkt werden. Dazu wird ein regulärer Ausdruck (pattern) verwendet, der nur die Zeichen *m* oder *w* zulässt.

Ein regulärer Ausdruck ist selbst eine Zeichenkette, die als Beschreibung von Mengen von Zeichenketten mithilfe eines eigenen, sehr komplexen und mächtigen Regelwerks dient. Reguläre Ausdrücke stellen also eine Art Filterkriterium für Texte dar, indem der jeweilige reguläre Ausdruck in Form des Musters mit einem gegebenen Text aus der XML-Datei abgeglichen wird. Im XML Schema wird auf diese Weise ein Filter für die Obermenge der Zeichenketten angegeben.

Im Anschluss an die Definition eines Teilnehmers mit dem neuen Datentyp *Geschlecht* wird in Listing 4.73 die Teilnehmerliste definiert, die aus mindestens einem Teilnehmer (da *minOccurs* nicht angegeben wurde, ist die Mindestzahl automatisch 1) bis unendlich vielen Teilnehmern (*maxOccurs="unbounded"*) besteht.

Abschließend wird noch definiert, dass die Liste der Vorkenntnisse aus mindestens einer Vorkenntnis besteht. Auch hier ist keine obere Grenze der Vorkenntnisse vorgesehen.

Damit ist das XML-Format der Seminare ausreichend beschrieben. Beachten Sie bitte, dass die Reihenfolge der Definitionen beliebig ist. Sie können beispielsweise die Teilnehmerliste einsetzen, bevor das Element der Teilnehmerliste definiert wurde.

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="dozent"/>
  <xs:element name="name"/>
  <xs:element name="vorkenntnis"/>
```

Listing 4.73: Schemadatei zur Prüfung des XML-Dokuments

```
  <xs:element name="seminar">
    <xs:complexType mixed="true">
      <xs:sequence>
        <xs:element ref="name" />
        <xs:element ref="dozent" />
        <xs:element ref="teilnehmerliste" />
      </xs:sequence>
      <xs:attribute name="id" type="xs:string" use="required" />
    </xs:complexType>
  </xs:element>

  <xs:element name="seminare">
    <xs:complexType>
      <xs:sequence><xs:element ref="seminar" /></xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="teilnehmer">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="name" />
        <xs:element ref="vorkenntnisse" minOccurs="0" />
      </xs:sequence>
      <xs:attribute name="geschlecht" type="gesch" use="required" />
      <xs:attribute name="id" type="xs:integer" use="required" />
    </xs:complexType>
  </xs:element>

  <xs:simpleType name="gesch">
     <xs:restriction base="xs:string">
      <xs:pattern value="m|w"/>
     </xs:restriction>
  </xs:simpleType>

  <xs:element name="teilnehmerliste">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="teilnehmer" maxOccurs="unbounded" />
      </xs:sequence>
    </xs:complexType>
```

Listing 4.73: Schemadatei zur Prüfung des XML-Dokuments (Forts.)

```
  </xs:element>

  <xs:element name="vorkenntnisse">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="vorkenntnis" maxOccurs="unbounded" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Listing 4.73: Schemadatei zur Prüfung des XML-Dokuments (Forts.)

Genau wie bei der Prüfung der XML-Beispieldatei anhand der DTD muss auch das Schema auf eine XML-Datei angewendet werden. In diesem Beispiel wird wieder die XML-Datei aus Listing 4.63 verwendet. Ein Hinweis auf das zugehörige Schema innerhalb der XML-Datei ist nicht notwendig.

In Listing 4.74 erfolgt nun die Prüfung auf Gültigkeit der *beispiel.xml* gegen das Schema *seminare.xsd*. Nach dem Laden der XML-Datei in das Document Objekt Model erfolgt über die bereits im DOM definierte Methode *schemaValidate* die Prüfung unter Angabe des Schemas. Diese Prüfung liefert genau wie die DTD-Prüfung im Erfolgsfall ein *TRUE* und andernfalls ein *FALSE* zurück.

```
<?php $doc=DOMDocument::load('beispiel.xml'); ?>
<html><body>
<?php
  if ($doc->schemaValidate('seminare.xsd')){
    echo 'Die Datei beispiel.xml ist gültig.';
  }
  else{
    echo 'Die Datei beispiel.xml NICHT ist gültig!';
  }
?>
</body></html>
```

Listing 4.74: Validieren der XML-Datei gegen das Schema

Sie erkennen auch hier, dass die eigentliche Prüfung leicht zu programmieren ist; die bereits von PHP 5 vorgegebenen Klassen und Objekte mit ihren Methoden nehmen Ihnen die Arbeit des Parsens und der Prüfung ab. Die Prüfung im Schema zu definieren ist allerdings bereits komplex genug. Andererseits wird das XML Schema die DTD-Sprache mittelfristig ablösen, da das Schema selbst von einem XML-Parser interpretiert werden kann.

Profitipp

Wenn Sie die DTD oder das Schema komplett von Hand erstellen, ist dies zwar eine gute Übung, jedoch ist der Aufwand bis zu einem erfolgreichen Ergebnis gerade bei größeren Dokumenten sehr hoch. Alternativ dazu existieren bereits Tools, mit denen Sie eine DTD oder ein XML Schema aus einer bestehenden XML-Datei generieren können. Dies ist auf den ersten Blick unlogisch, jedoch können die DTD oder das Schema als Vorlage für eine Überarbeitung des Regelwerks dienen. Eine solche Generierung können Sie unter anderem auf der Homepage *http://www.hitsw.com/xml_utilites/* vornehmen.

Transformation von XML zu anderen Ausgaben: XLST

Es wurde bereits erwähnt, dass das XML-Format insbesondere dem flexiblen Datenaustausch zwischen größeren Applikationen, beispielsweise aus dem B2B-Bereich, dient. Ein Datenstamm wird also über das XML-Format von einem Datenmodell in ein anderes Datenmodell überführt. Dies ist meist deshalb nötig, weil die verschiedenen Applikationen verschiedene ER-Modelle ihrer Datenbanken hinterlegt haben.

Um eine Konvertierung zwischen XML-Formaten vorzunehmen, wurde eine eigene Konvertierungssprache entworfen, die XLST (Extensible Stylesheet Language Transformation). Die XSL ist eine in XML definierte Familie von Transformationssprachen zur Definition von Layouts für XML-Dokumente. Man trennt also den reinen Datenstamm in der XML-Datei von dessen Darstellung in einer XSL-Datei. XLST ist eine Transformationssprache wie auch XSL-FO (Extensible Stylesheet Language – Formating Objects). Mit einer XSL-FO-Beschreibung können Sie beispielsweise eine XML-Datei in eine PDF-Datei zum Druck aufbereiten. Wie die Daten aus der XML-Datei dargestellt werden sollen, bestimmen Sie in der separaten XSL-FO-Beschreibung, ähnlich, wie Sie in einem separaten Schema die Gültigkeit der Daten definieren.

Zum Einstieg wird in diesem Kapitel eine Transformation in ein anderes XML-Format vorgenommen. Da der Befehlssatz der HTML-Sprache als Untermenge von XML aufgefasst werden kann, können Sie die gegebene XML-Datei aus Listing 4.63 unter Verwendung einer XLST-Transformationsdatei direkt in eine HTML-Datei umwandeln. Das Prinzip der Transformation wird in Abbildung 4.17 verdeutlicht.

Listing 4.75 zeigt eine solche Transformationsdatei. Sie enthält neben dem darzustellenden HTML-Rumpf von den Elementen *<html>* bis *</html>* zusätzliche Anweisungen, wie die Daten aus der XML-Datei in die HTML-Ausgabe eingepflegt werden sollen. Eine solche Anweisung lautet beispielsweise *<xsl:value-of select="seminare/seminar/name"/>*. Dadurch wird in dem Name des ersten Seminars in der XML-Datei extrahiert und an diese Stelle platziert. Auf Attribute eines XML-Elements können Sie durch ein vorangestelltes „@“ zugreifen. So gibt *<xsl:value-of select="seminare/seminar/@id"/>* die ID des Seminars zurück.

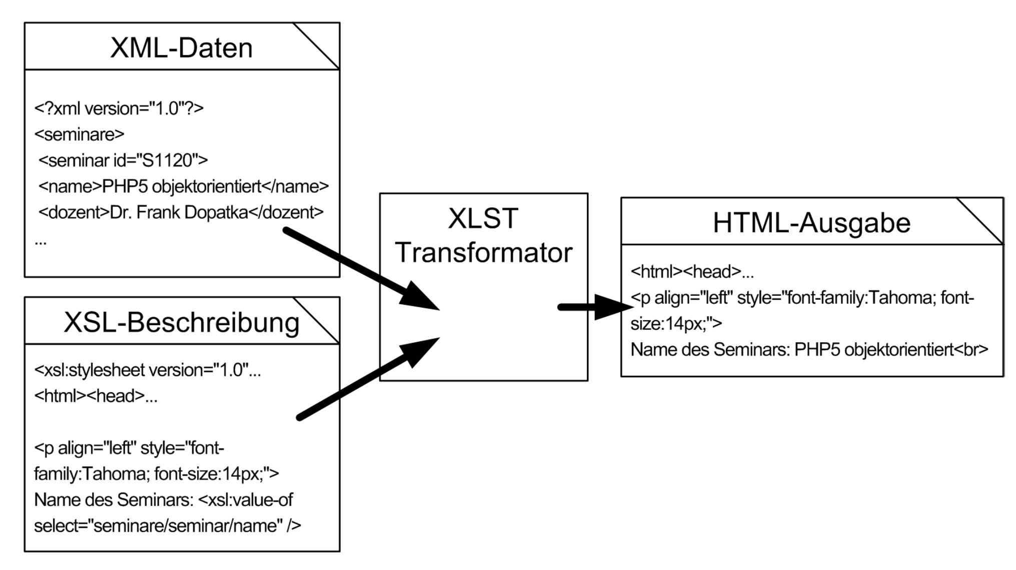


Abbildung 4.17: Prinzip einer XSL-Transformation in ein HTML-Dokument

> **Hinweis**
>
> Die XSL-Transformationsdatei aus Listing 4.75 ist nur für ein einzelnes Seminar gedacht, um die Übersichtlichkeit des Beispiels zu wahren.

Da es sich bei XLST um eine eigene vollständige Programmiersprache handelt, können Sie auch Schleifen verwenden. Im Beispiel wird eine Schleife verwendet, um die Namen aller Teilnehmer auszugeben. Die Zeile *<xsl:for-each select="seminare/seminar/teilnehmerliste/teilnehmer">* nimmt Bezug auf den Pfad im Baum des XML-Dokuments und erinnert gleichzeitig an die *foreach*-Schleife von PHP. Die Namen der Teilnehmer werden dann in eine HTML-Auflistung innerhalb der HTML-Tags *
...
* eingebettet.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<html><head></head><body>
<p align="left" style="font-family:Tahoma; font-size:14px;">
  Name des Seminars: <xsl:value-of select="seminare/seminar/name"/><br/>
  ID des Seminars: <xsl:value-of select="seminare/seminar/@id"/><br/>
  Dozent des Seminars: <xsl:value-of select="seminare/seminar/dozent"/>
  <br/>
</p>
<p align="left" style="font-family:Tahoma; font-size:14px;">
  Teilnehmer:
```

Listing 4.75: Transformationsdatei für die XML-Datei nach HTML – beispiel.xls

```
  <ul style="font-family:Tahoma; font-size:14px;">
    <xsl:for-each select="seminare/seminar/teilnehmerliste/teilnehmer">
      <li><xsl:value-of select="name"/></li>
    </xsl:for-each>
  </ul>
</p>
</body></html>
</xsl:template>
</xsl:stylesheet>
```

Listing 4.75: Transformationsdatei für die XML-Datei nach HTML – beispiel.xls

Die Aufbereitung in ein HTML-Dokument wurde mit Listing 4.75 beschrieben. Nun fehlt noch die Anwendung der Transformation. Da es sich sowohl bei der *beispiel.xml* als auch bei der *beispiel.xls* um XML-Dateien handelt, werden beide zunächst als DOM-Objekt geladen. Dies geschieht in der Testdatei aus Listing 7.48.

Die Transformation erfolgt über einen Transformationsprozessor, der – wie üblich in der Objektorientierung – zunächst über den *new*-Operator erzeugt werden muss. Sie erhalten dann ein Prozessorobjekt *$proc* mit einem Satz von eigenen Methoden/Diensten, die dieser Prozessor bereitstellt. Eine dieser Methoden, *importStylesheet*, besteht darin, eine Transformationsbeschreibung in den Prozessor zu laden. Ein Beschreibungsobjekt steht in der Objektreferenz *$xsl* bereits zur Verfügung. Die Methode *transformToXML* transformiert die im Eingabeparameter angegebene Datei nun in ein anderes Format. In unserem Fall wird dabei eine HTML-Datei erzeugt, die dann direkt ausgegeben wird.

```
<?php
  $xsl=DomDocument::load('beispiel.xsl');
  $xml=DomDocument::load('beispiel.xml');
  $proc=new XsltProcessor();
  $proc->importStylesheet($xsl);
  print $proc->transformToXML($xml);
?>
```

Listing 4.76: HTML-Ausgabe

In der beschriebenen Kombination von XML- und XSL-Datei führt dies zu der folgenden Ausgabe:

Name des Seminars: PHP5 objektorientiert

ID des Seminars: S1120

Dozent des Seminars: Dr. Frank Dopatka

Teilnehmer:

- *Ulf Schmidt*
- *Clair Grube*

- *Ulla Hansen*
- *Franz Streber*
- *Hans Wurst*

Auf diese Art und Weise können Sie verschiedene Transformationen, beispielsweise zur Ausgabe auf mobile Endgeräte oder in andere XML-Formate zum Import der Daten in andere Systeme, definieren und halten den Datenstamm in Form der XML-Datei nur einmalig vor.

Hinweis

Selbstverständlich kann im Rahmen dieses Buches keine vollständige Beschreibung der Sprachen DTD, Schema und XSL(T) erfolgen. Zu diesem Zweck existieren bereits eine Vielzahl von Literatur sowie Internetquellen. In diesem Buch sollen Sie nur die prinzipiellen Technologien mit grundlegenden Beispielen zum Einstieg sowie den Einsatzzweck dieser Technologien kennen lernen.

4.5 Ein Web Service in PHP

Eine besondere, verteile Art der objektorientierten Programmierung stellen Web Services dar. Ein Web Service ist ein Dienst, also eine Methode bzw. eine Funktionalität, die zumeist in einem Intranet bereitgestellt wird.

Web Services orientieren sich an der serviceorientierten Architektur (SOA) und vereinen verteilte und objektorientierte Programmierstandards, wobei sie insbesondere betriebswirtschaftliche Lösungen fokussieren. Eine Anwendung kann einen Web-Service über ihren Uniform Resource Identifier (URI) eindeutig identifizieren und ihren Dienst ausführen, der mit seinem Ergebnis antwortet.

Ein besonderes Augenmerk liegt dabei auf der Kommunikation zwischen dem Nutzer des Dienstes und dessen Anbieter. Diese Kommunikation erfolgt über SOAP (Simple Object Access Protocol). Dabei handelt es sich um ein Netzwerkprotokoll, mit dem Daten zwischen unterschiedlichen Systemen ausgetauscht und entfernte Prozeduraufrufe, so genannte Remote Procedure Calls (RPC), durchgeführt werden können. Dabei müssen die verschiedenen Systeme nicht in derselben Programmiersprache implementiert sein. So kann ein in PHP geschriebener Web Service prinzipiell von einer Java-Anwendung genutzt werden und umgekehrt.

Listing 4.77 zeigt einen ersten in PHP implementierten Web Service. Dabei soll die einfache Funktion *addiere* im Intranet bereitgestellt werden, die zwei Eingabeparameter erhält und einen Wert als Ergebnis zurückgibt.

Dazu wird zunächst ein neuer SOAP-Server instantiiert, der als Parameter die URI erhält, unter der der Server zu finden ist. Im zweiten Schritt wird die zu veröffentliche Funktion dem Server hinzugefügt. Abschließend wird eine eingehende Dienstanfrage über die Methode *$server->handle()* des Serverobjekts verarbeitet und die Antwort automatisch an den Aufrufer des Dienstes zurückgesendet.

```
<?php
  function addiere($s1, $s2){
    return $s1 + $s2;
  }

  $server = new SoapServer(NULL,
                       array('uri' => "http://localhost/webservice/"));
  $server->addFunction('addiere');
  $server->handle();
?>
```

Listing 4.77: Der erste Web Service in PHP

Nun fehlt noch der Aufrufer, der den bereitgestellten Dienst verwendet. In Listing 4.78 wird ein Clientobjekt erstellt, das sich mit dem Server verbindet. Dann werden die Eingabeparameter für den aufzurufenden Dienst definiert. Diese Parameter werden als *SoapParam* bezeichnet. Die Ursache für diese Deklaration liegt darin, dass die aufgerufene Methode nicht unbedingt in PHP implementiert sein muss. Sie können von dem PHP-Client aus auch prinzipiell auf einen in Java oder C# implementierten Web Service zugreifen. Der *SoapParam* ist ein Datentyp, der in SOAP definiert und daher ein von einer speziellen Programmiersprache unabhängiger Datentyp ist.

Im dritten Schritt wird die *addiere*-Methode vom Client aus aufgerufen und damit ein SOAP-Request an den PHP-Server gesendet. Der verarbeitet die Anfrage und gibt das Ergebnis zurück, das in der Variablen *$result* gespeichert und letztlich ausgegeben wird.

```
<?php
  $client = new SoapClient(NULL,
  array(
    "location" => "http://localhost/webservice/server.php",
    "uri" => "urn:xmethodsTestServer",
    "style" => SOAP_RPC,
    "use" => SOAP_ENCODED
  ));

  $parameters = array(
    new SoapParam('10', 's1'),
    new SoapParam('20', 's2'));

  $result = $client->__call(
  "addiere",
  $parameters,
    array(
```

Listing 4.78: Der erste Dienstbenutzer

```
      "uri" => "urn:xmethodsTestServer",
      "soapaction" => "urn:xmethodsTestServer#addiere"
    ));
  echo $result;
?>
```

Listing 4.78: Der erste Dienstbenutzer (Forts.)

Um den Programmierer des Clients nicht mit der umständlichen Definition der SOAP-Parameter zu belästigen, kann die Definition auch serverseitig erfolgen und dem Client zur Verfügung gestellt werden. Die Implementierung des Servers aus Listing 4.77 wird davon nicht beeinflusst.

Stattdessen wird dem Server eine WSDL-Datei beiseite gestellt. Bei der Web Service Description Language handelt es sich um eine auf XML basierende Sprache zur plattform-, programmiersprachen- und protokollunabhängigen Beschreibung von Web Services. Eine solche Datei ist in Listing 4.79 dargestellt.

Die an den Testserver gestellte Anfrage zur Addition besitzt zwei Parameter, deren Datentypen mit *<part name='sum1' type='xsd:float'/>* als Gleitkommazahlen definiert werden. Ebenso wird der Rückgabewert des Dienstes als Gleitkommazahl definiert. Die Verbindung der Eingabe- und Ausgabeparameter zur Methode *addiere* erfolgt durch die Definition der Eingangs- und Ausgangsnachricht im Element *<operation name='addiere'>*.

Im nächsten Schritt werden noch Parameter für das Nachrichtenformat und das verwendete Kommunikationsprotokoll gesetzt. So erfolgt die Kommunikation über einen entfernten Prozeduraufruf unter Verwendung des HTTP-Protokolls.

Im Anschluss daran wird der Funktionsaufruf *addiere* beschrieben. Der Uniform Ressource Name (URN) beschreibt einen dauerhaften, ortsunabhängigen Bezeichner für eine Ressource. Hier handelt es sich bei der Ressource um den Dienst der Addition, der über die URN eindeutig angesprochen werden kann. Die Ein- und Ausgabeparameter werden RPC-encoded übertragen. Dabei handelt es sich um eine sehr einfache Art der Kodierung, die als Teil des SOAP-Protokolls spezifiziert ist.

Abschließend wird noch der URL des Serverdienstes spezifiziert.

```
<?xml version ='1.0' encoding ='UTF-8' ?>
<definitions name='TestServer'
  xmlns:tns=' http://localhost/webservice/server.wsdl'
  xmlns:soap='http://schemas.xmlsoap.org/wsdl/soap/'
  xmlns:xsd='http://www.w3.org/2001/XMLSchema'
  xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
  xmlns:wsdl='http://schemas.xmlsoap.org/wsdl/'
  xmlns='http://schemas.xmlsoap.org/wsdl/'>

<message name='addiereAnfrage'>
```

Listing 4.79: Die WSDL-Datei für die Funktion der Addition

```
  <part name='sum1' type='xsd:float'/>
  <part name='sum2' type='xsd:float'/>
</message>
<message name='addiereAntwort'>
  <part name='Result' type='xsd:float'/>
</message>
<portType name='TestServerPortType'>
  <operation name='addiere'>
    <input message='tns:addiereAnfrage'/>
    <output message='tns:addiereAntwort'/>
  </operation>
</portType>
<binding name='TestServerBinding' type='tns:TestServerPortType'>
  <soap:binding style='rpc'
    transport='http://schemas.xmlsoap.org/soap/http'/>
  <operation name='addiere'>
    <soap:operation soapAction='urn:xmethodsTestServer#addiere'/>
    <input>
      <soap:body use='encoded' namespace='urn:xmethodsTestServer'
        encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'/>
    </input>
    <output>
      <soap:body use='encoded' namespace='urn:xmethodsTestServer'
        encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'/>
    </output>
  </operation>
</binding>
<service name='TestServerService'>
  <port name='TestServerPort' binding='TestServerBinding'>
    <soap:address location='http://localhost/webservice/server.php'/>
  </port>
</service>
</definitions>
```

Listing 4.79: Die WSDL-Datei für die Funktion der Addition (Forts.)

Nun stellt sich die Frage, wie viel Aufwand man noch betreiben will, um eine einfache Addition durchzuführen. Tatsächlich ist die XML-Beschreibung der WSDL-Datei extrem komplex, was jedoch auf ihre Plattformunabhängigkeit zurückzuführen ist. Andererseits existieren bereits Hilfsmittel für den Umgang mit WSDL-Dateien, wie der WSDL-Editor von Altova (*http://www.altova.com/de/xmlspy/wsdl-editor.html*).

Der Vorteil der WSDL-Datei liegt darin, dass der Client diese Datei direkt auslesen und interpretieren kann, wie es in Listing 4.80 dargestellt ist. Im Anschluss an die Interpreta-

tion der Datei kann der Client den Dienst wie einen gewöhnlichen, lokalen Methodenaufruf verwenden.

```
<?php
  $client = new SoapClient('http://localhost/webservice/server.wsdl');
  $result = $client->addiere(10, 20);
  echo $result;
?>
```

Listing 4.80: Der vereinfachte Client mit Zugriff auf die WSDL-Datei

4.6 Neuerungen in PHP 5.3 und Ausblick

Seit dem 30. Juni 2009 ist die momentan aktuelle Version PHP 5.3 verfügbar, die in die Version 1.72 des XAMPP-Pakets (*http://www.xampp.de*) integriert wurde. Zunächst ist anzumerken, dass die neue Unterversion von PHP natürlich keine umfassende Änderung der Sprache darstellt. Die letzte große Erweiterung der Sprache im Hinblick auf Objektorientierung wurde mit der Version 5.0 vom 13. Juli 2004 durch die neue Zend Engine II vorgenommen. Die Grundzüge der Umsetzung objektorientierter Prinzipien wurde bereits in diesem Kapitel ausführlich vorgestellt.

An dieser Stelle werden nun lediglich die Neuerungen skizziert, die zwischen der seit November 2006 bestehenden Version 5.2 und der aktuellen Version 5.3 von PHP bestehen. Wenn Sie sich erstmals mit PHP beschäftigen, werden Sie die Unterschiede der beiden Versionen kaum wahrnehmen.

Insbesondere wurden einige Details und einige Konzepte der Objektorientierung verbessert. Das Kapitel endet schließlich mit einem Ausblick auf Neuerungen der PHP-Version 6.0, deren Erscheinungstermin bislang jedoch noch unbekannt ist.

4.6.1 Namensräume für größere Softwaremodule

Ein Name identifiziert eine Klasse, wie einen „Kunden“ oder ein „Auto“. Um in größeren Softwarekomponenten eine eindeutige Zuordnung beibehalten zu können, ist neben dem Namen einer Klasse der entsprechende Kontext zu beachten. Dieser Kontext wird als Namensraum bezeichnet. Die Beschreibung geschieht in PHP durch die "/"-Notation.

Auch ein Dateisystem ist ein Namensraum, in dem Sie Dateien anordnen, sortieren und wiederfinden können. Wie auch Ordner in einem Dateisystem Unterordner enthalten können, kann jeder Namensraum auch wiederum Namensräume enthalten. So einsteht eine hierarchische, baumförmige Struktur mit einer Wurzel als Ausgangspunkt, Knoten und Blättern. Namensräume werden auch dazu verwendet, Konflikte bei der Namensvergabe zu verhindern.

Beim Erstellen einer großen Anwendung können Programmierteams unter der Benutzung von Namensräumen große Programmpakete schreiben, ohne dass die neu eingeführten Namen in Konflikt zu anderen Namen stehen. Im Unterschied zu einer Anwen-

dung ohne Namensräume wird dabei nicht der ganze Name einer Klasse neu eingeführt, sondern nur ein neues Blatt im gesamten Pfad des Namensraums eingefügt.

Damit ist es seit PHP 5.3 möglich, einen Namen in unterschiedlichen Namensräumen konfliktfrei zu verwenden, auch wenn der gleiche Name in der gleichen Übersetzungseinheit vorkommt. Dadurch, dass er in unterschiedlichen Namensräumen erscheint, ist jeder Name eindeutig auf Grund des Pfads im Namensraum zugeordnet.

Namensräume können in Verbindung mit einer Unterteilung der Anwendung in Komponenten durch ein UML-Paketdiagramm dargestellt werden, das die Struktur der (zu erstellenden) PHP-Anwendung meist auf einer höheren Ebene skizziert.

Bei der Analyse und Modellierung von Geschäftsprozessen werden Pakete oft benutzt, um fachlich zusammengehörende Teile des Modells zusammenzufassen. Ein Paketdiagramm stellt eine Übersicht der Geschäftsprozesse dar, beispielsweise eine Auftrags- oder eine Rechnungsverwaltung.

Abbildung 4.18 skizziert eine kleine ERP-Anwendung (Enterprise Ressource Planning) mit einer Kunden- und Artikelverwaltung, wobei jedem Kunden Rechnungen zugeordnet werden können, die wiederum aus Rechnungspositionen bestehen, die ihrerseits einen Bezug zu verkauften Artikeln besitzen.

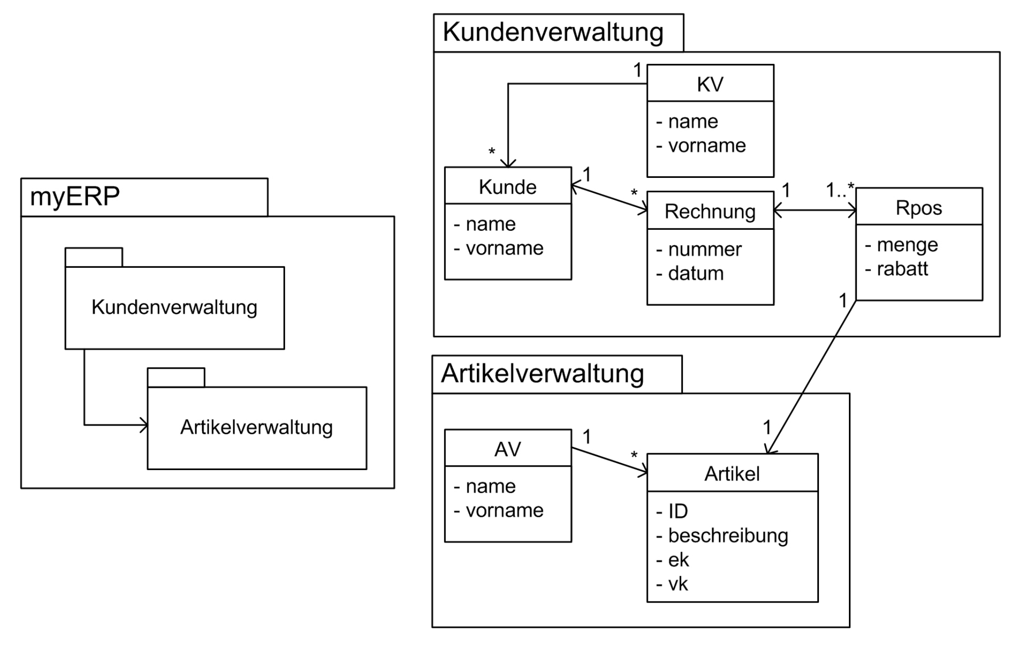


Abbildung 4.18: UML-Paketdiagramm der Kunden- und Artikelverwaltung

Der Kern des ERP-Systems besteht also aus einer Kunden- und Artikelverwaltung, die ihrerseits wieder aus einer Vielzahl von Klassen und eigener Benutzeroberfläche bestehen können.

Aus Sicht der Modellierung könnte aus der Kundenverwaltung noch eine Rechnungsverwaltung extrahiert werden, sodass drei Komponenten der Anwendung entstehen würden. Zur Vereinfachung des Beispiels beschränken wir uns jedoch auf diese beiden Namensräume.

In diesem Beispiel wird die Testklasse ausnahmsweise zuerst verfasst. Listing 4.81 zeigt zunächst die Einbindung der beiden Verwaltungsklassen, für die der Klassenlader aus Listing 4.8 weiterverwendet werden kann.

Im Anschluss daran wird jeweils ein Objekt der Kunden- und der Artikelverwaltung erzeugt. Die PHP-Klassen befinden sich in den Unterverzeichnissen *myERP\kv* bzw. *myERP\av*.

```
<html><body>
<?php
  require_once("classloader.inc.php");
  // Verwendung einer Klasse aus unterliegendem Namensraum:
  $kv = new myERP\kv\Kundenverwaltung();
  $av = new myERP\av\Artikelverwaltung();
?>
</body></html>
```

Listing 4.81: Testklasse start.php

Da sich die Baumtiefe der Namensräume über mehrere Ebenen erstrecken kann, können Sie Aliasbezeichnungen für die Verwendung von Pfaden verwenden. In Listing 4.82 wird dazu der neue PHP-Befehl *use* verwendet in Kombination mit den Aliasnamen, die in diesem Beispiel *x* und *y* lauten. So können die Verwaltungsobjekte immer noch präzise angesprochen werden.

```
<html><body>
<?php
  require_once("myERP/kv/kundenverwaltung.inc.php");
  require_once("myERP/av/artikelverwaltung.inc.php");
  // Alias-Definition:
  use myERP\kv as x;
  use myERP\av as y;
  // Vewendung mit Alias
  $kv = new x\Kundenverwaltung();
  $rv = new y\Artikelverwaltung();
?>
</body></html>
```

Listing 4.82: Testklasse start.php mit verkürzten Namensräumen

Es fehlen noch die Implementierungen der Verwaltungsklassen, die in den Listings 4.85 und 4.86 skizziert werden. Wichtig ist dabei, dass diese Klassen als erste Anweisung stets den Bezug zur Wurzel im Namensraum über den Befehl *namespace* besitzen.

Diese und ähnliche Notationen sind auch in anderen objektorientierten Programmiersprachen geläufig. In Java lautet sie *package myERP.kv;* und in C# *namespace myERP.kv.*

```
<?php
  namespace myERP\kv;

  class Kundenverwaltung{
    private $kundenliste=null;

    public function __construct(){
      echo('Eine Kundenverwaltung wird erzeugt...<br>');
    }
  }
?>
```

Listing 4.83: Hauptklasse der Kundenverwaltung im Unterverzeichnis myERP/kv

Eine Kunden- oder Artikelverwaltung besteht natürlich nicht nur aus jeweils einer Klasse. Vielmehr können hinter den Verwaltungsklassen ganze Klassengeflechte mit vielen Zeilen Quellcode verborgen sein, die für den Anwender der Verwaltungsklasse jedoch nur mäßig interessant sind. Die Namensräume sorgen also gleichzeitig für eine bessere Strukturierung der Softwarekomponenten, bei denen einzelne Zugangspunkte – hier die Verwaltungsklassen – für den Benutzer dieser Komponenten bereitgestellt werden. Da die Hilfsklassen in verschiedenen Ordnern im Dateisystem gespeichert sind, die über die Namensräume gezielt angesprochen und aufgelöst werden können, werden Namenskollisionen vermieden.

```
<?php
  namespace myERP\av;

  class Artikelverwaltung{
    private $artikelliste=null;

    public function __construct(){
      echo('Eine Artikelverwaltung wird erzeugt...<br>');
    }
  }
?>
```

Listing 4.84: Hauptklasse der Artikelverwaltung im Unterverzeichnis myERP/av

Da die Verwaltungsklassen lediglich skizziert wurden, um sich auf die Funktionsweise der Namensräume zu konzentrieren, lautet die Testausgabe schlicht:

Eine Kundenverwaltung wird erzeugt...

Eine Artikelverwaltung wird erzeugt...

Weitere Informationen zu den Namensräumen können Sie in der Onlinedokumentation von PHP unter *http://www.php.net/manual/de/language.namespaces.php* einsehen.

4.6.2 Softwaremodule in PHP-Pakete bündeln: Phar

Eine weitere Neuerung in PHP 5.3 wurde aus dem Java-Umfeld übernommen. Bereits im vorherigen Kapitel wurde begründet, dass große Softwaremodule in Pakete aufgeteilt werden sollten, die sich in verschiedenen Ordnern befinden und über die Namensräume angesprochen werden können. Genau diese Unterteilung existiert auch in Java. Genauso wie in Java kann ein einzelnes Paket auch in PHP eine Vielzahl von Klassen enthalten, sodass sich eine große Sammlung von Dateien ergibt. Selbst bei einer Unterteilung in verschiedenen Ordnern wird das Projekt dadurch unübersichtlich.

Aus diesem Grund lassen sich mehrere Klassen in Java zu einem Archiv zusammenfassen, das die Dateiendung *.jar* erhält als Abkürzung für Java Archive. Ein solches Archiv beinhaltet eine Vielzahl von kleinen kompilierten *.class*-Dateien in einer Verzeichnisstruktur, die im ZIP-Format komprimiert wurden. Zusätzlich wird in einem Unterordner *META-INF* eine Textdatei *manifest.mf* erstellt, die Metainformationen zu dem Archiv enthält, wie Inhalte, Versionsnummer oder auch Hersteller des Archivs.

In der Version 5.3 hat PHP analog dazu das *.phar*-Format eingeführt, was PHP-Archiv bedeutet. Dabei stellt sich zunächst die Frage, wie Sie eine solche *.phar*-Datei anlegen. In Listing 4.85 wird aus allen Dateien der Kundenverwaltung aus Abbildung 4.18 ein PHP-Archiv angelegt.

```
<html><body>
<?php
  $phar=new Phar('kv.phar');
  $phar->buildFromDirectory('myERP\kv');
?>
</body></html>
```

Listing 4.85: Erstellung eines PHP-Archivs

Es ist jedoch wahrscheinlich, dass der Aufruf des Skripts in der Fehlermeldung *Fatal error: Uncaught exception 'UnexpectedValueException' with message 'creating archive "kv.phar" disabled by INI setting' in...* endet. Dies liegt daran, dass Sie das Schreiben in ein Archiv zunächst in der Konfigurationsdatei *php.ini* erlauben müssen, die sich im PHP-Ordner der Installation befindet. Dazu müssen Sie den Kommentar im Eintrag *phar.readonly* entfernen und *phar.readonly = Off* setzen. Nach einem Neustart des Apache-Webservers können Sie dann das Skript erneut ausführen. Das Skript erzeugt nun eine neue Datei *kv.phar* im Ordner der Datei aus Listing 4.85, die nicht ZIP-komprimiert ist. Sie

beinhaltet alle Dateien des Zielordners im hinteren Teil der PHAR-Datei, der mit zusätzlichen PHP-Funktionen im vorderen Teil zusammengefügt wurde.

Listing 4.86 zeigt, wie Sie die Dateien des PHAR-Archivs textuell wieder auslesen können. Die Referenz *$item* zeigt dabei auf ein Dateiobjekt im Archiv, dessen Name Sie über den Methodenaufruf *getFilename()* auslesen können. Der Name *my.phar* ist dabei ein Alias für das Archiv, damit Sie unabhängig vom eigentlichen Namen des Archivs auf dessen Inhalte zugreifen können.

```
<html><body>
<?php
  $phar=new Phar('kv.phar',0,'my.phar');
  foreach($phar as $item){
    var_dump($item->getFilename());
  }
?>
</body></html>
```

Listing 4.86: Auslesen der Archivinhalte

Abschließend müssen Sie noch wissen, wie Sie auf die Kundenverwaltungsklasse innerhalb des Archivs zugreifen können. Dazu müssen Sie zunächst wieder einen Alias definieren, mit dessen Hilfe Sie dann über eine besondere Pfaddefinition die Datei aus dem Archiv inkludieren. Dann können Sie wie gewohnt über den entsprechenden Namensraum ein neues Objekt dieser Klasse anlegen.

```
<html><body>
<?php
  $phar=new Phar('kv.phar',0,'my.phar');
  require_once('phar://my.phar/Kundenverwaltung.inc.php');
  use myERP\kv as x;
  $kv = new x\Kundenverwaltung();
?>
</body></html>
```

Listing 4.87: Zugriff auf eine Klasse im Archiv

4.6.3 PHP in Verbindung mit Windows-Servern

Da PHP aus der Open-Source-Gemeinde stammt und in Verbindung mit dem stabilen und weit verbreiteten Webserver Apache seine Berühmtheit erlangt hat (Stichwort LAMP-Server und XAMPP zur vereinfachten Installation), mangelte es lange Zeit an der Unterstützung und Integration in die Microsoft-Windows-Welt. Obwohl PHP auf Windows-Betriebssystemen schon lange problemlos installiert werden konnte (Stichwort WAMP-Server), ist die wesentlich höhere Verbreitung im Linux-Umfeld lange auch durch die höhere Verfügbarkeit von Linux-Servern zu erklären gewesen.

Obwohl Microsoft mit den Active Server Pages und deren Integration in die .NET-Welt und in den Internet Information Server (IIS) eine eigene Strategie verfolgt, die sogar in Konkurrenz zu PHP steht, verstärkt Microsoft seit dem IIS 6 die Anbindung zur Sprache PHP. Dies geschieht über eine FastCGI-Schnittstelle (Common Gateway Interface), die man kostenlos herunterladen kann.

Zusätzlich dazu bietet Microsoft finanzielle Unterstützung für das PHP-Projekt und hat angekündigt, PHP zukünftig auf allen Windows-Serverplattformen zu unterstützen.

Seit PHP 5.3 wurde die Unterstützung von einigen PHP-Funktionen, die zuvor nur auf Linux-Servern zuverlässig funktionierten, im Microsoft-Umfeld verbessert. Dazu gehören unter anderem *checkdnsrr()* und *checkmxrr()* zur Überprüfung von Domain- und URL-Angaben sowie *link()* zur Erzeugung eines absoluten Links im Dateisystem des Servers.

4.6.4 Änderung im Late Static Binding

Bei den PHP-Versionen vor 5.3 sorgte die Verwendung von Klasseneigenschaften öfters für Verwirrung bei den Programmierern. Betrachten Sie den folgenden Quellcode, der eine Klasseneigenschaft definiert und diese in einer Unterklasse überschreibt. Laut den Regeln der Objektorientierung müsste die Get-Methode den neuen, überschriebenen Wert 22 ausgeben.

```
<?php
class Test{
  static protected $x=7;
  public static function getX(){
    return self::$x;
  }
}
class SpeziellerTest extends Test{
  static protected $x=22;
}
?>
<html><body>
<?php
  var_dump(SpeziellerTest::getX());
?>
</body></html>
```

Listing 4.88: Statische Klassenattribute und deren Vererbung

Dies geschieht jedoch nicht, stattdessen erfolgt die Ausgabe des Wertes 7. Die Ursache dafür liegt darin, dass das Schlüsselwort *self* in der Get-Methode bereits beim Übersetzen des Programms durch den Namen der Klasse – in diesem Fall also durch den Namen der Oberklasse – ersetzt wird. Das Schlüsselwort *this* dürfen Sie hier nicht verwenden, da es sich auf eine Objektreferenz bezieht und bei der Abfrage der Klasseneigenschaft ja noch gar kein Objekt dieser Klasse existieren muss.

Seit PHP 5.3 können Sie nun den Rückgabewert durch den Aufruf *return self::$x;* ermitteln. Das neue Schlüsselwort *static* sorgt dafür, dass erst zur Laufzeit der Name der Klasse ermittelt wird, die die Klasseneigenschaft ausliest.

Analog dazu wurde die Methode *__call* (Kap. 4.1.4.) für den Zugriff auf Eigenschaften um eine weitere Methode ergänzt, die *__callStatic* genannt wird. Diese Methode wird dann ausgeführt, wenn Sie eine nicht existierende statische Methode aufgerufen haben.

4.6.5 Neue und verbesserte Funktionen

Mit der neuen Version 5.3 wurden auch einige neue, nichtobjektorientierte Einzelfunktionen ergänzt, die in Tabelle 4.4 aufgeführt werden.

Name	Funktion
quoted_printable_encode()	konvertiert einen so genannten „quoted-printable"-String nach RFC2045 in einen 8-Bit-String, z. B. „schön" in „sch=C3=B6n"; die entsprechende *decode*-Methode *quoted_printable_decode()* existierte bereits in älteren PHP-Versionen
preg_filter()	führt ein Suchen und Ersetzen in einem Datenstamm unter Verwendung eines regulären Ausdrucks durch
parse_ini_string()	liest eine Zeichenkette mit Konfigurationsangaben ein und gibt die Konfigurationsdaten als assoziatives Feld zurück; die Konfigurationsangaben müssen in einem Format vorliegen, wie es in der *php.ini* verwendet wird nach der Notation *Parametername = Wert*
lcfirst()	wandelt den ersten Buchstaben einer Zeichenkette in einen Kleinbuchstaben um; die Funktion zum Großschreiben des ersten Buchstabens *ucfirst()* existiert bereits in vorherigen PHP-Versionen
array_replace()	ersetzt Elemente in einem Datenfeld durch andere Elemente, die von außen übergeben werden
array_replace_recursive()	wie *array_replace()*, funktioniert jedoch auch für Felder, die wiederum Felder enthalten

Tabelle 4.4: Neue Funktionen in PHP 5.3

Außerdem wurden einige bestehende Funktionen in PHP 5.3 um weitere Parameter ergänzt, die eine verbesserte Handhabung der Funktionen ermöglichen. Die Parameter sind alle optional und wurden so eingefügt, dass Skriptquellcodes für ältere PHP-Versionen unverändert ausgeführt werden können. Die Erweiterungen betreffen die Funktionen

- *array_reduce* wendet eine übergebene Funktion iterativ bei den Elementen eines Feldes an, sodass das Feld auf einen einzigen Wert reduziert wird
- *clearstatcache* löscht den Status-Cache des Skripts
- *copy* kopiert eine Datei

- *fgetcsv* liest eine Zeile von der Position des Dateizeigers aus und prüft auf Daten im CSV-Format (Comma Separated Values)
- *getimagesize* ermittelt die Größe einer Bilddatei
- *ini_get_all* liest Konfigurationsoptionen vollständig aus
- *nl2br* fügt vor allen Zeilenumbrüchen einer Zeichenkette HTML-Zeilenumbrüche ein
- *round* rundet eine Fließkommazahl
- *stream_context_create* erzeugt einen Datenstrom
- *strstr* findet das erste Auftreten einer Zeichenkette in einer anderen Zeichenkette

Eine genauere Beschreibung der neuen Möglichkeiten dieser Funktionen können Sie in der Onlinereferenz unter *http://www.php.net/manual/en/* nachlesen. Geben Sie dort im Suchfeld den Namen der Funktion ein. Die Neuerungen werden innerhalb der Dokumentation im Changelog festgehalten. Bitte beachten Sie, dass die deutsche Version der Onlinereferenz nicht immer auf dem neuesten Stand ist!

4.6.6 Ausblick auf PHP 6

Auch wenn der Erscheinungstermin des nächsten großen Versionssprungs auf PHP 6 noch unbekannt ist, werden bei den Entwicklern von PHP und in Foren bereits einige notwendige Erweiterungen diskutiert, die in der aktuellen Version 5.3 noch nicht enthalten sind.

Die erste geplante Erweiterung besteht in der Einführung des Unicode-Zeichensatzes. Dieser Zeichensatz ist bereits bei anderen objektorientierten Sprachen wie Java und dem .NET-Framework weit verbreitet. Der zentrale Unterschied zu einem landestypischen ASCII-Zeichen besteht darin, dass ein Unicode-Zeichen aus 2 Bytes statt aus einem Byte besteht.

Unicode bzw. das Universal Character Set (UCS) ist ein international genormter Standard nach ISO 10646, in dem für jedes Schriftzeichen aller bekannten Schriften ein digitaler Code festgelegt wird. Das Ziel ist es dabei, die Verwendung unterschiedlicher und inkompatibler Zeichensätze in verschiedenen Ländern zu beseitigen. Unicode wird laufend um weitere Zeichen ergänzt.

Die *ereg*-Befehl aus dem bisherigen PHP-Wortschatz soll in PHP 6 entfallen. Mit *ereg* können Sie eine Zeichenkette unter Berücksichtigung der Groß- und Kleinschreibung gegen einen regulären Ausdruck prüfen und ggf. eine Umformatierung des Ausdrucks vornehmen, sofern eine Übereinstimmung vorliegt. Der reguläre Ausdruck wird ebenfalls als Zeichenkette definiert. Der Quellcode aus Listing 4.89 konvertiert ein Datum *$date* im ISO-Format (JJJJ-MM-TT) in das in Deutschland typische Format TT.MM.JJJJ.

```
<?php
if (ereg ("([0-9]{4})-([0-9]{1,2})-([0-9]{1,2})", $date, $regs)) {
    echo "$regs[3].$regs[2].$regs[1]";
```

Listing 4.89: Umwandlung einer Datumsdarstellung mittels ereg-Befehl

```
} else {
    echo "Ungültiges Datum!";
}
?>
```

Listing 4.89: Umwandlung einer Datumsdarstellung mittels ereg-Befehl (Forts.)

Da PHP ursprünglich als einfach zu handhabende, leichte Skriptsprache konzipiert war, haben sich einige Vorgehensweisen eingebürgert, aus denen sich Sicherheitslücken im PHP-Code ergeben. Bereits in den aktuellen PHP-Versionen wurde die Verwendung dieser Vorgehensweisen standardmäßig in der *php.ini* abgeschaltet, um die Sicherheit der Sprache zu erhöhen. Ein unvorsichtiger Administrator oder Programmierer kann diese Vorgehensweisen durch Änderung der *php.ini* jedoch wieder aktivieren. Dies soll in PHP 6 nicht mehr möglich sein. Zu diesen Funktionen zählt *register_globals*. Bei Aktivierung kann man auf zuvor definierte Variablen direkt zugreifen, die mittels HTTP-Get, HTTP-Post oder über einen Cookie vom Client übertragen werden.

```
<?php
  if ($loggedin){
    fpassthru ("/portal/index.html");
  }
?>
```

Listing 4.90: Globale Registrierung register_globals=on

Stattdessen soll bereits jetzt aus Sicherheitsgründen der Zugriff dediziert über ein von PHP vordefiniertes Datenfeld erfolgen, wie es in Listing 4.91 dargestellt wird. Manipulationen bei der Kommunikation zwischen Client und Server werden dadurch erschwert (jedoch nicht unmöglich!).

```
<?php
  if($_COOKIE[loggedin]){
    // kann nur von einem Cookie stammen
    $good_login = 1;
    fpassthru ("/portal/index.html");
  }
?>
```

Listing 4.91: Globale Registrierung register_globals=off

Ein weiteres Element ist die Einstellung *magic_quotes* der *php.ini* seit der vierten PHP-Version, die in PHP 6 wieder entfernt werden soll. Dabei werden allen Request-Variablen an den nötigen Stellen mit Backslashes versehen, was als „Escapen" bezeichnet wird. Das entspricht der Anwendung der Funktion *addslashes()* auf alle Variablen. In der praktischen Anwendung waren die Probleme dieser Idee jedoch größer als deren Nutzen. So kann die Einstellung auf jedem Server anders konfiguriert sein. Außerdem können sich Backslashses „vermehren", wenn Daten öfter zwischen Server und Client hin und her

versendet werden. Außerdem bieten einige Datenbankfunktionen eigene Escaping-Methoden, was zu doppeltem Escapen und damit zu ungültigen Anweisungen für die Datenbank führen kann. Außerdem darf beim Schreiben in Dateien und bei Ausgaben zum Internetbrowser generell kein Escapen vorgenommen werden, da auch dies zu fehlerhaften Darstellungen führen würde.

Ebenso kann die Einstellung des *safe_mode* in zukünftigen PHP-Versionen nicht mehr verändert werden, wodurch der Safe Mode von PHP stets aktiviert bleibt. Der Safe Mode verhindert beispielsweise, dass ein Kunde eines Shared Hosters auf die Dateien eines anderen Kunden über ein PHP-Skript zugreifen kann.

Schließlich soll noch eine PHP-Erweiterung mit Namen *mime_magic* entfallen. Mit den Multipurpose Internet Mail Extensions (MIME) wird der Aufbau von Internetnachrichten festlegt. Ferner findet MIME Anwendung bei der Deklaration von Inhalten in verschiedenen Internetprotokollen wie HTTP und ermöglicht den Austausch von Informationen über den Typ der übermittelten Daten zwischen Sender und Empfänger.

Im Gegenzug zu *mime_magic* soll die bereits jetzt existierende PHP-Bibliothek *fileinfo* mit MIME-Support ausgebaut werden.

Zusammenfassend ist zu sagen, dass die Neuerungen in PHP 6 im Vergleich zu PHP 5 wahrscheinlich nicht so umfangreich sein werden wie der Umstieg von PHP 4 auf PHP 5, mit dem ja durch die Objektorientierung ein völlig neues Programmierparadigma in die Sprache aufgenommen wurde. Bei jeder neuen Version ist zu erkennen, dass die Kompatibilität zu bereits erstellten PHP-Skripten nach Möglichkeit über mehrere Versionen gewahrt bleiben soll, um den Erfolg der Sprache auch in Zukunft zu sichern.

5 Projektpraxis

Im dritten Kapitel dieses Buches haben Sie etwas über die Vorgehensweise der Abwicklung von (größeren) Projekten in einem iterativ-inkrementellen agilen Prozess erfahren. Die UML als gemeinsame Sprache aller Projektbeteiligten spielt dabei sowohl als Notation als auch als Richtschnur von der Projektidee bis zur Implementierung und Dokumentation eine zentrale Rolle.

Das vierte Kapitel hat gezeigt, wie die Konzepte der Objektorientierung mit PHP 5 umgesetzt werden können, indem jedes einzelne Konzept in einem kurzen, unabhängigen Beispiel mit UML skizziert und dann in PHP 5 implementiert wurde.

Die Inhalte in Kapitel 5.1 verbinden nun die beiden vorherigen Kapitel, indem ein einziges Projekt von seiner Analyse bis hin zu den ersten implementierten Prototypen verfolgt wird. Im Anschluss daran werden noch Regeln für „guten" PHP-Quellcode vorgestellt sowie eine Reihe von Tools, die bei der alltäglichen Arbeit mit PHP-Projekten behilflich sind und auch die Qualität der erstellten PHP-Anwendung positiv beeinflussen.

5.1 Das Fallbeispiel der Depotverwaltung

Dieses Fallbeispiel zeigt die ersten Iterationen eines Softwareprojekts, bei dem ein virtueller Auftraggeber, die „RAUB-Bank", einen Onlinedienst zur Verwaltung der Aktiendepots seiner Anleger plant. Die ersten Gespräche mit dem IT-Manager der Bank werden in Kapitel 5.1.1 skizziert.

Dem folgt eine fachliche, objektorientierte Analyse der Problemstellung mit dem Ziel einer ersten fachlichen Modellierung in Kapitel 5.1.2. Das Kapitel 5.1.3 dargestellte objektorientierte Design führt dann zu einem technischen Modell, das die Grundlage für die Implementierung der ersten Prototypen bildet. Das Prototyping in PHP 5.3 erfolgt dann iterativ-inkrementell, wobei das Konzept der testgetriebenen Entwicklung (Kap. 3.2.5) angewendet wird.

Nachdem die ersten Prototypen umgesetzt wurden, werden zum Abschluss des Fallbeispiels die nächsten Schritte in Kooperation mit dem Auftraggeber besprochen.

5.1.1 Die Idee des Auftraggebers

Gerade in Zeiten der Wirtschaftskrise mit niedrigen Zinsen auf Konten möchte die IT-Abteilung der RAUB-Bank ihren Kunden einen neuen Dienst kostenlos zur Verfügung stellen. Da die Aktienkurse sehr stark gefallen sind, möchte unser Auftraggeber möglichst schnell ein System bereitstellen, mit dem ein Kunde der Bank sein Aktiendpot ver-

walten kann. Ein Depot besteht dabei aus Beständen von verschiedenen Aktien, die der Kunde zuvor gekauft hat.

Der Anleger als Kunde der Bank soll neben den aktuellen Kursen jederzeit eine Übersicht seiner Gewinne bzw. Verluste erhalten können. Zusätzlich dazu soll ein einfacher Vergleich der Zinsen aus den Aktiengewinnen mit Zinssätzen von Festgeld- und Spareinlagen möglich sein. Im Gegensatz zu anderen Depotverwaltungsprogrammen sollen auch Gewinne aus Dividenden berücksichtigt werden können, die Aktiengesellschaften üblicherweise jährlich an die Aktionäre im Anschluss an die Hauptversammlungen ausschütten.

Die zu erstellende PHP-Anwendung soll sich aber (zunächst) nicht direkt mit einem Aktiendepot des Anlegers verbinden. Stattdessen soll jeder Anleger seinen Aktienbestand und jede Transaktion zunächst manuell eingeben. Dadurch soll ein Kunde der Bank sich auch ein Musterdepot anlegen können, um in den Aktienhandel „hineinzuschnuppern“. Die Bank erhofft sich dadurch die Gewinnung von Neukunden.

Eine Transaktion ist neben dem Kauf auch ein Verkauf eines Aktienbestands. Ebenso wird der Erhalt einer Dividende als zusätzliche Zahlung an den Anleger als *Transaktion* bezeichnet und vom System durch manuelle Eingabe erfasst.

Die aktuellen Kurse sollen aber nicht vom Anleger manuell eingegeben werden. Stattdessen sollen diese Kurse aktuell von einer existierenden Homepage eingelesen werden können. Solche Homepages existieren nach den Aussagen des Auftraggebers für jede Aktie, die man kaufen kann.

5.1.2 Die objektorientierte Analyse

Nach dem ersten Gespräch mit unserem Auftraggeber werden nun die gewünschten Funktionen erstmals definiert und genauer beleuchtet. Dies geschieht unter Verwendung einzelner grafischer Anwendungsfalldiagramme, von denen einzelne Funktionen in einer textuellen Schablone genauer beschrieben werden.

Typische Abläufe, die sich unser Auftraggeber im System vorstellt, werden im Anschluss daran mit Aktivitätsdiagrammen festgehalten. Diese Abläufe beschreiben Interaktionen des zukünftigen Benutzers mit der zu erstellenden Anwendung.

Gewünschte Funktionen mit Anwendungsfalldiagrammen

Im ersten Schritt der Analyse wird der Auftraggeber der RAUB-Bank darum gebeten, darzustellen, welche Hauptfunktionen er in der zu erstellenden Anwendung sieht. Welche Funktionen ständen groß auf der Verpackung, wenn man das Produkt in einer Schachtel kaufen könnte?

Seine Antworten werden in Abbildung 5.1 als erster grafischer Anwendungsfall zusammengefasst. Der Auftraggeber will

- Aktienbestände verwalten
- Aktienbestände archivieren
- die aktuellen Kurse automatisch ermitteln

Damit jeder Benutzer dies kann, muss er sich zunächst am System anmelden. In den ersten Phasen der Umsetzung soll das Anmeldesystem mit der Benutzerverwaltung jedoch nicht im Vordergrund stehen.

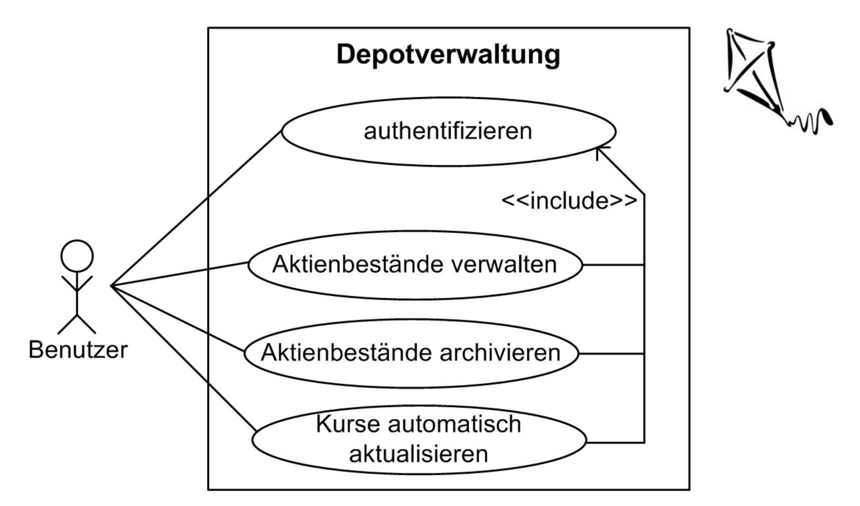


Abbildung 5.1: Grafischer Anwendungsfall auf Drachenebene

Im nächsten Schritt werden die Funktionen genauer hinterfragt. Konzentrieren Sie sich als Systemanalytiker dabei vor allem auf weich formulierte Oberbegriffe. Dies sind in unserem Fall „verwalten" und „archivieren". Was ist damit genau gemeint?

Als Antwort auf diese Frage müssen Sie Ihren Auftraggeber dazu bringen, die Funktionen detaillierter zu beschreiben. Oft ist es auch hilfreich, einen (zukünftigen) Benutzer der Anwendung zu befragen, was er sich unter dem Dienst vorstellt und erhofft. Dadurch gelangt man von der Wolken- bzw. Drachenebene hin zu der Ebene des Meeresspiegels, die einzelne Funktionen beschreibt, die man später als Menüeinträge in der Anwendung wiederfinden sollte.

In unserem Fall bedeutet das Archivieren das Laden und Speichern der aktuellen Aktienbestände in einer Datenbank. Diese Datenbank soll nachträglich austauschbar sein; die Anwendung darf sich also nicht auf einen Datenbankhersteller so fokussieren, dass ein Wechsel der Datenbank nur mit hohem Aufwand verbunden wäre.

Unser Auftraggeber sieht die Verwaltung der Aktienbestände als Kernfunktion der Anwendung. Der Benutzer soll einen neuen Aktienbestand anlegen, indem er Informationen zur Aktie und zum ersten Kauf (Anzahl der gekauften Aktien, Kaufdatum und Kurs der Aktie) in die Anwendung eingibt.

Zusätzlich soll der Benutzer zu einem späteren Zeitpunkt Aktien desselben Typs zu einem bestehenden Bestand nachkaufen können. Die Daten des n-ten Kaufs werden dann in den Bestand integriert.

Ebenso sollen meist jährlich gezahlte Dividendenzahlungen der Aktiengesellschaften in den Bestand aufgenommen werden können. Die Dividenden werden aufaddiert und erhöhen den Ertrag der Aktie. Wenn eine Dividende gezahlt wurde, kann sie dem Anle-

ger nicht mehr weggenommen werden, während der Kurs der Aktie ja sinken kann. Eine Dividende ist demnach ein garantierter Gewinn.

Jeder Aktienbestand kann auch verkauft werden. In den ersten Prototypen genügt es, wenn man nur den gesamten Aktienbestand verkaufen kann. Dabei werden das Datum des Verkaufs und der Aktienkurs erfasst, zu dem der Bestand verkauft wurde. Zusätzlich fallen Verkaufsgebühren an.

Dies bringt die Diskussion auf das Gebührenmodell. Unser Auftraggeber betont, dass nur bei jedem Kauf und Verkauf bei seiner Bank Gebühren fällig sind. Das Depot verursacht also keine laufenden Kosten, die beispielsweise jährlich abgerechnet werden. Für zukünftige Gebührenmodelle sollte die Anwendung jedoch in diese Richtung erweiterbar sein.

Als letzte Funktion nennt der Auftraggeber die Übersicht über den gesamten Aktienbestand. Hier soll der Benutzer seine Bestände sowie die statistischen Informationen sehen können. Diese werden zunächst nur textuell präsentiert.

Abbildung 5.2 zeigt detaillierter die gewünschte Funktionalität der Anwendung als Anwendungsfall auf der Ebene des Meeresspiegels. Da die Anzahl der Funktionen noch überschaubar ist, werden die Verwaltung und Archivierung in einem einzigen Diagramm dargestellt.

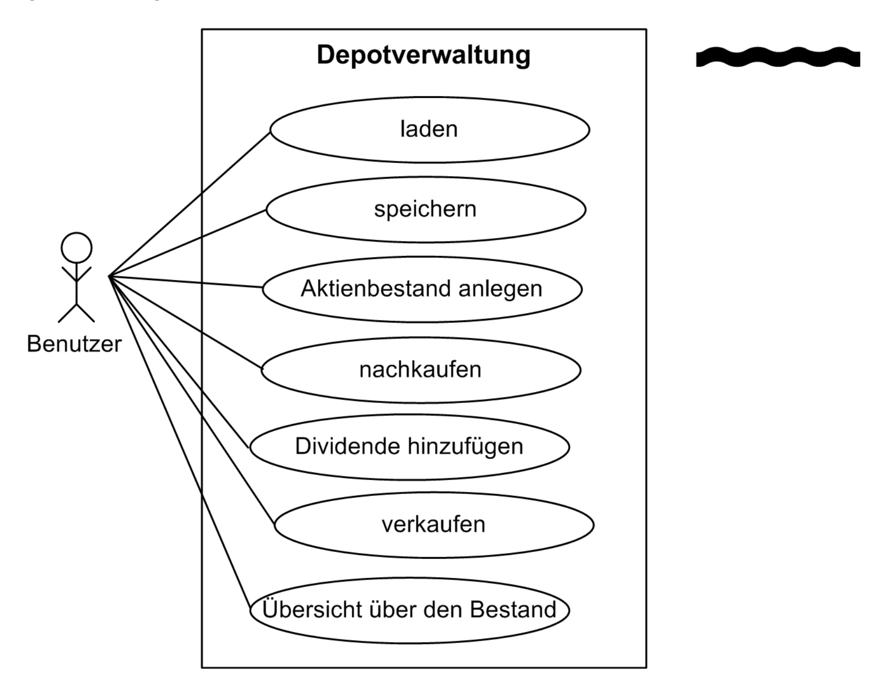


Abbildung 5.2: Grafischer Anwendungsfall auf Ebene des Meeresspiegels (Benutzersicht)

Bei einer größeren Anzahl von Funktionen versuchen Sie bitte, diese weiter zu gruppieren und die Gruppierung auf der Drachenebene darzustellen. Die jetzige Drachenebene würde dann als reine Managementsicht auf die Wolkenebene befördert. Der Meeresspie-

gel würde dann aus einer Vielzahl von Anwendungsfalldiagrammen bestehen, die jeweils eine Gruppe von Funktionen realisieren.

Die in Abbildung 5.2 dargestellten Funktionen werden nun weiter hinterfragt und spezifiziert. Wie stellt sich der Auftraggeber das Anlegen eines neuen Aktienbestands vor? Was ist dazu alles an Daten notwendig? Wie soll das Anlegen aus Sicht des Benutzers ablaufen? Zu jedem Anwendungsfall sollte man als nächsten Schritt eine textuelle Anwendungsfallschablone (Abb. 3.33) und/oder ein Aktivitätsdiagramm erstellen.

Die Inhalte der textuellen Anwendungsfallschablone und des Aktivitätsdiagramms überlappen sich leicht, sodass Sie überlegen können, auf eines der beiden Verfahren zu verzichten. Zur Übung werden im Folgenden beide Verfahren durchgegangen.

Dabei wird in Abbildung 5.3 zunächst die Anwendungsfallschablone für den Use-Case *nachkaufen* erstellt.

Aktien nachkaufen
Ziel: Benutzer hat Aktien nachgekauft. **Vorbedingung:** Benutzer ist angemeldet und hat bereits einen Aktienbestand mit Aktien des gewünschten Typs. **Nachbedingung Erfolg:** Benutzer besitzt weitere Aktien desselben Typs in seinem Aktienbestand. **Nachbedingung Fehlschlag:** Mitteilung an Benutzer, das seine Eingaben fehlerhaft waren oder die Aktien aus technischen Gründen nicht in den Bestand aufgenommen werden konnten. **Akteure:** Benutzer der Depotverwaltung **Auslösendes Ereignis:** Benutzer möchte nachgekaufte Aktien in das Verwaltungsprogramm einpflegen. **Beschreibung:** 1. Bestand auswählen mit Aktien des Typs, die nachgekauft werden sollen. 2. „Nachkaufen“ auswählen. 3. Daten des Nachkaufs einpflegen (Menge, Kaufdatum, Kurs, Gebühren) und bestätigen. 4. Erfolgsmeldung über das Einpflegen in den Bestand. **Erweiterung:** 3a. direkter Check der Eingaben des Benutzers auf Gültigkeit **Alternativen:** 1a. neuen Bestand anlegen, wenn noch keine Aktien dieses Typs existieren 4a. Meldung bei fehlerhaften Eingaben und Korrekturmöglichkeit

Abbildung 5.3: Textueller Anwendungsfall zum „Nachkaufen“ von Aktien

Achten Sie dabei insbesondere auf eine saubere Definition der Vorbedingung. Was muss erfüllt sein, damit der Anwendungsfall ausgeführt werden kann? Wodurch wird der Anwendungsfall ausgelöst? Die zweite Antwort ergibt das auslösende Ereignis.

Die Definition der Fehlschläge aus fachlicher (nicht aus technischer!) Sicht und die Reaktion darauf sind von ebenso großer Bedeutung wie die Beschreibung (Wie kommt man auf dem kürzesten Weg zu einer erfolgreichen Ausführung?) und die Erweiterungen bzw. Alternativen.

Während die Beschreibung den Primärfluss darstellt, der in frühen Prototypen zu realisieren ist, sind die Erweiterungen Hinweise auf mögliche optionale oder „Nice to have"-Funktionen. Alternativen beschreiben hingegen eher leichte Verzweigungen in der Beschreibung und damit alternative Wege, um zum Erfolg zu gelangen.

Meinung

Sie erkennen, dass die erste Phase sehr textlastig ist. Es gilt hier vor allem, Begriffe, Aufgaben und Funktionen zu definieren und eine gemeinsame Sprache mit dem Auftraggeber zu finden, der meist aus einem anderen Fachgebiet stammt. Sie als Analytiker müssen dabei in den Gesprächen und Workshops die „richtigen" Fragen stellen. Eine kreative Atmosphäre mit verschiedenen Personen, also verschiedenen Sichtweisen, ist dabei hilfreich. Versuchen Sie auch herauszufinden, welche Funktionen nicht zwingend in den ersten Schritten des Projekts notwendig sind. Ein Ausfüllen der Schablonen im stillen Kämmerlein und insbesondere ein Copy-Paste-Ausfüllen erzeugt zwar Papier, aber nur einen geringen Mehrwert!

Gewünschte Abläufe mit Aktivitätsdiagrammen

Ähnlich wie die Beschreibung, die Erweiterungen und Alternativen der textuellen Anwendungsfälle stellen die Aktivitätsdiagramme Abläufe im Geschäftsprozess dar, die man auch als Workflows bezeichnet. Die im Folgenden dargestellten Diagramme befinden sich vorwiegend auf der Wasserspiegelebene und reichen leicht in die Fischebene herunter, die aber noch von einem Benutzer nachvollzogen werden kann.

In Abbildung 5.4 wird das Vorgehen eines Benutzers beschrieben, der einen neuen Aktienbestand anlegen will. Der Benutzer kommuniziert dabei mit dem zu erstellenden Programm zur Depotverwaltung.

Nachdem der Benutzer den Dienst *neuen Bestand anlegen* ausgewählt hat, möchte unser Auftraggeber dem Benutzer eine Eingabemaske zur Verfügung stellen. Dort sind Angaben zur Aktie sowie die Daten des ersten Kaufs einzugeben. Zu den Angaben zur Aktie gehören insbesondere

- der Name der Aktie
- die ISIN (International Securities Identification Number), eine zwölfstellige Buchstaben-Zahlen-Kombination nach ISO 6166, die eine weltweit eindeutige Identifikation für ein Wertpapier darstellt; man kann diese Nummer also als Primärschlüssel betrachten

Die wichtigsten Daten zum ersten Kauf, die den vorhandenen Aktienbestand ausmachen, sind

- das Datum des Kaufs
- die Anzahl der gekauften Aktien
- deren Kurs, zu dem die Aktien gekauft wurden
- Gebühren, die bei dem Kauf angefallen sind

Die notwendigen Daten erzeugen ein neues Aktienbestandsobjekt, nachdem die eingegebenen Daten auf Gültigkeit geprüft worden sind. Dieses Objekt wird dann in der Depotverwaltung festgehalten, worüber der Benutzer abschließend informiert wird.

Profitipp

Definieren Sie stets zunächst das Primärszenario, das der Beschreibung der textuellen Schablone entspricht. Ihnen mag dies zunächst zu trivial erscheinen, aber bei der Besprechung mit dem Auftraggeber und den Benutzern ergeben sich dadurch viele neue Erkenntnisse. Bedenken Sie: Die Diagramme dienen in erster Linie nicht dem Selbstzweck, sondern als Diskussionsgrundlage!

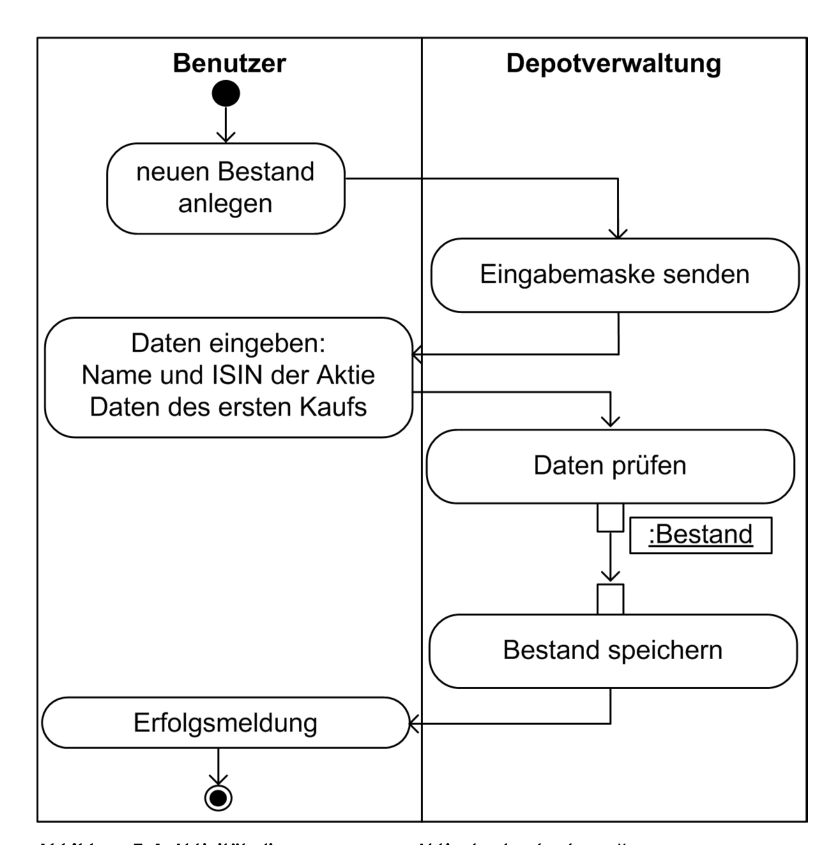


Abbildung 5.4: Aktivitätsdiagramm „neuer Aktienbestand anlegen"

Wenn mehrere Bestände angelegt worden sind, soll der Benutzer einen Bestand zur weiteren Verarbeitung auswählen können. Dieser Vorgang ist in Abbildung 5.5 dargestellt. Nach der Aufforderung des Benutzers zur Auswahl eines Bestands wird zunächst eine Liste des Gesamtbestands angezeigt. Zusätzlich soll an dieser Stelle zu jedem Bestand seine aktuelle Bilanz erscheinen.

Mit dem Begriff der Bilanz ist in diesem Fall keine Bilanz im betriebswirtschaftlichen Sinne gemeint. Unser Auftraggeber möchte dem Kunden an dieser Stelle die Statistik mit allen Gewinnen und Verlusten der Bestände sowohl in Euro, als auch in Prozent präsentieren. Wenn ein Aktienbestand noch nicht verkauft wurde, also noch im Besitz des Anlegers ist, sollen die aktuellen Kurse der Börse als Referenzdaten verwendet werden. Wie Sie dies realisieren, überlässt der Auftraggeber Ihnen als Entwickler.

Aus der Liste der Bestände kann der Benutzer nun einen Bestand auswählen, der noch nicht verkauft wurde. Den gewählten Bestand merkt sich das Verwaltungsprogramm. Somit kann der Benutzer den gewählten Bestand verkaufen, neue Aktien nachkaufen oder eine Dividende hinzufügen.

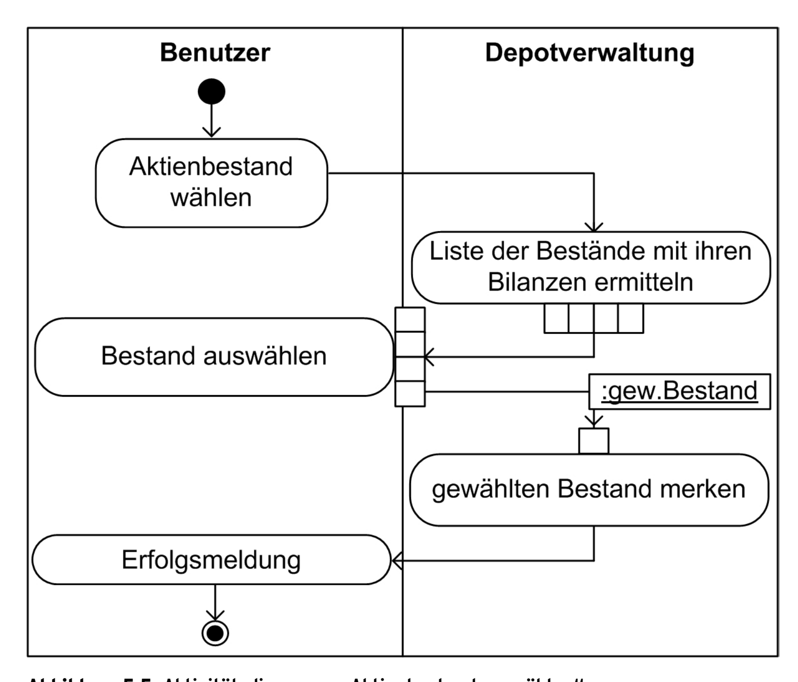


Abbildung 5.5: Aktivitätsdiagramm „Aktienbestand auswählen“

Abbildung 5.6 zeigt das Aktivitätsdiagramm, um eine Dividendenzahlung in Euro zu einem bereits gewählten Aktienbestand hinzuzufügen. Dies geschieht wiederum über eine Eingabemaske der Depotverwaltung. Die Eingabe des Benutzers wird geprüft und die Daten des Bestands werden aktualisiert. Wie üblich, endet das Szenario mit einer Erfolgsmeldung an den Benutzer.

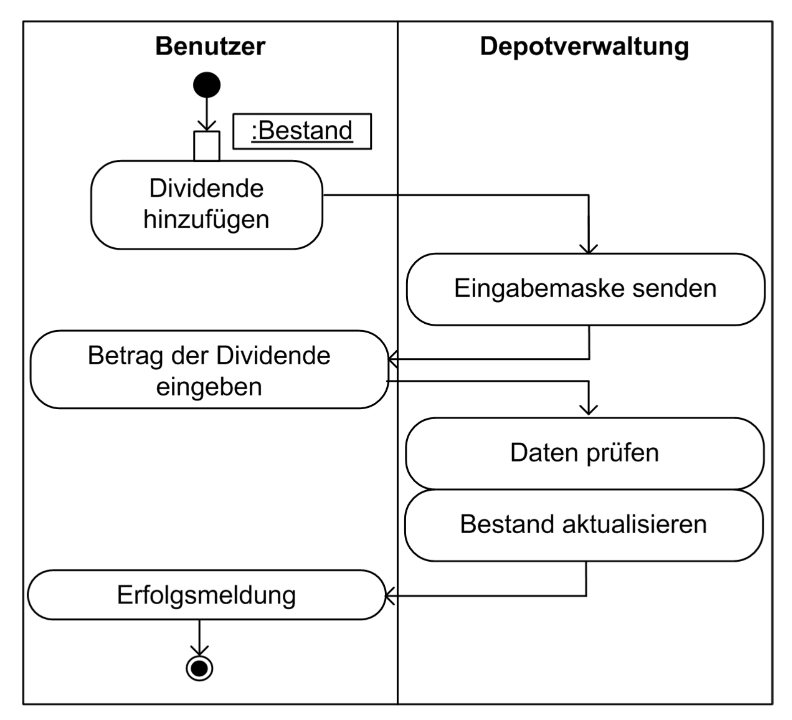


Abbildung 5.6: Aktivitätsdiagramm „Dividende eingeben“

Als letztes Szenario wird in Abbildung 5.7 beschrieben, wie zu einem ausgewählten Aktienbestand neue Aktien hinzugekauft werden sollen. In der Sprache der Aktionäre wird dies übrigens als erfolgreiche Kauforder, ein Verkauf von Aktien als durchgeführte Verkauforder bezeichnet. Um mehr Aktien desselben Typs in den Bestand aufzunehmen, müssen die Daten des neuen Kaufvorgangs über eine Eingabemaske erfasst werden. Dabei handelt es sich wie bereits beim Anlegen des Bestands um

- das Datum des Kaufs
- die Anzahl der gekauften Aktien
- deren Kurs, zu dem die Aktien gekauft wurden
- Gebühren, die bei dem Kauf angefallen sind

Nach der Prüfung wird der aktuelle Bestand dann wieder in der Depotverwaltung gespeichert.

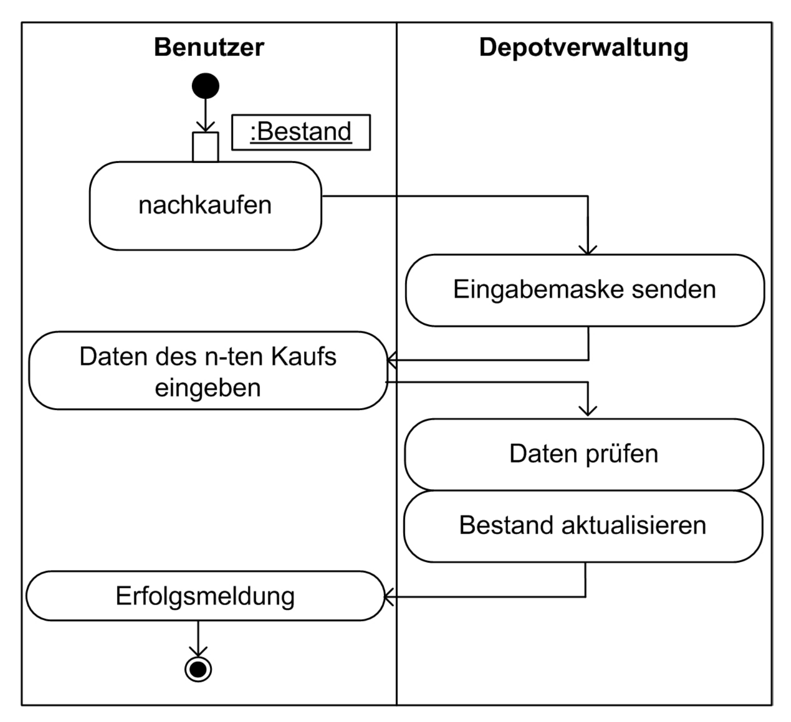


Abbildung 5.7: Aktivitätsdiagramm „Aktien zu vorhandenem Bestand nachkaufen“

In diesem Zusammenhang stellt sich die Frage, wie das Aktualisieren des Bestands aussieht. Sollen alle einzelnen Kaufvorgänge im Bestand unabhängig voneinander festgehalten werden? Dies entspricht in der Implementierung eine Liste von Kaufvorgängen im Bestand. Überraschenderweise verneint unser Auftraggeber die Antwort und skizziert folgendes Beispiel:

Nehmen Sie an, Sie haben zum Zeitpunkt *D1 = 01.01.2010 Anz1 = 200* Aktien vom Typ A für *Kurs1 = 1.00 €/Stück* gekauft. Dann kauften Sie zum Zeitpunkt *D2 = 01.03.2010* nochmals *Anz2 = 100 Aktien* á *Kurs2 = 2.00 €/Stück*. Nun ist der Zeitpunkt *D3 = 01.06.2010* und Sie wollen wissen, wie viel Euro bzw. wie viel Prozent Zinsen Sie bis heute erwirtschaftet haben. Der Kurs der Aktie liegt jetzt bei *Kurs3 = 3.8 €/Stück*. Gebühren sollen nicht berücksichtigt werden.

Zum Zeitpunkt *D1* haben Sie 2 von 3 Anteilen an Ihrem heutigen Bestand gekauft und zum Zeitpunkt *D2* einen weiteren Anteil. Sie haben also 2/3 Ihres heutigen Bestandes zu je 1.00 €/Stück und 1/3 zu je 2.00 €/Stück. Grafisch würden Sie dabei vorgehen wie in Abbildung 5.8 dargestellt.

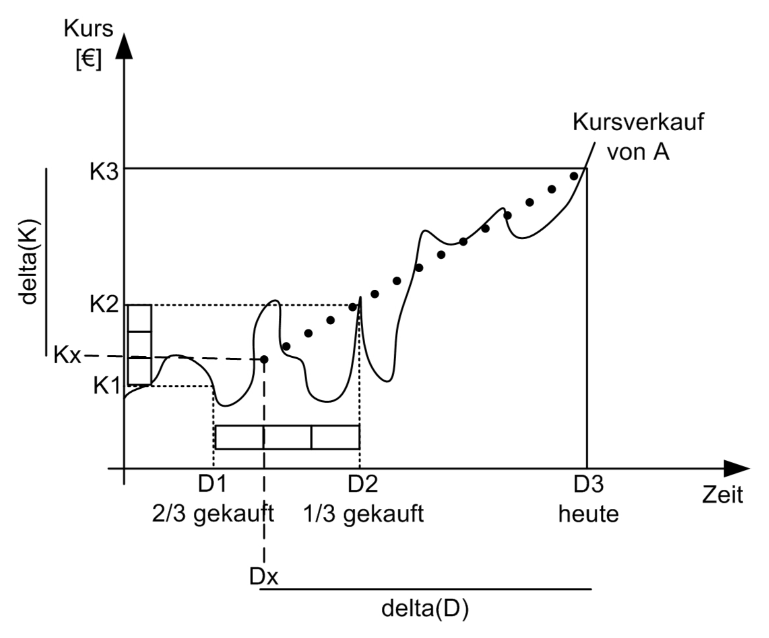


Abbildung 5.8: Berechnung eines virtuellen Aktienkaufs

Sie bilden den Mittelwert zwischen den beiden Datumswerten und auch zwischen den beiden Kursen unter Berücksichtigung der gekauften Anzahl an Aktien. Dies wird in der Mathematik als gewichteter arithmetischer Mittelwert bezeichnet. Der Mittelwert für das Datum *Dx* kann somit berechnet werden, indem man von *D1* aus den beim zweiten Kaufvorgang gekauften Anteil am jetzigen Bestand, also das Gewicht g, addiert. Das zu addierende Gewicht können Sie ermitteln mit der Formel *$g = $Anz2 / ($Anz1 + $Anz2).*

Zwischen *D1* und *D2* sind 59 Tage vergangen. Diese Anzahl der vergangenen Tage zwischen zwei Datumswerten können Sie übrigens mit der PHP-Funktion *strtotime* ermitteln, die als Parameter ein Datum als Zeichenkette erhält und einen UNIX-Zeitstempel zurückliefert. Über die Formel *$anzTage = (strtotime($D2)-strtotime($D1))/86400* können Sie dann die Anzahl der vergangenen Tage ermitteln. Um eine Anzahl an Tagen zum Datum *D1* zu addieren, können Sie bei der Implementierung die PHP-Funktion *date_add* verwenden. Diese benötigt ein Datumsobjekt als Eingabe, sodass Sie *D1* zunächst in ein Datumsobjekt umwandeln müssen. Als zweiten Parameter erwartet die Funktion eine speziell formatierte Zeichenkette, die die Anzahl der zu addierenden Tage enthält. Die Berechnung zur Ermittlung des neuen Datums *Dx* lautet also *$Dx = date_add(new DateTime($D1), new DateInterval('P'.round($g*$anzTage,0).'D')).*

Die entsprechende Berechnung für die Kurse gestaltet sich etwas einfacher, da Sie dort ja direkt mit Zahlenwerten arbeiten. Der gewichtete Mittelwert für den Kurs einer einzelnen Aktie *Kx* lautet *Kx = $K1 + $K1 * $g.*

Selbstverständlich gehört noch kein PHP-Code in die frühe Phase der objektorientierten Analyse. Sie können stattdessen auch eine mathematische Schreibweise zur Definition der Formeln verwenden.

Sinnvoll sind in dieser Phase allerdings das Aufstellen der notwendigen Berechnungen und deren Prüfung vom Auftraggeber. Ebenso ratsam ist es, die Berechnungen bereits sehr früh in PHP zu implementieren und die Ergebnisse zu verifizieren.

Spätestens jetzt wird der Auftraggeber Sie fragen, wie Sie den Aufwand für die Realisierung des Projekts bzw. bis zum ersten Prototyp einschätzen. Wie viel Zeit haben Sie bis jetzt in Meetings verbracht? Wie aufwändig war die Erstellung der Anwendungsfälle, die Einarbeitung in die Sprache des Auftraggebers und die Erstellung der Aktivitätsdiagramme?

Als Nächstes müssen Sie die Klassen und deren Verbindungen zueinander ermitteln, die Eigenschaften und Methoden festlegen, die Eingabemasken und die fachliche Logik implementieren.

Hinweis

Bevor Sie an dieser Stelle weiterlesen: Schätzen Sie den bisherigen Aufwand und den noch zu erstellenden Aufwand zur Umsetzung der Funktionen des Anwendungsfalldiagramms aus Abbildung 5.2. Wie viele Mannstunden würden Sie für eine Realisierung ohne die Behandlung von Fehlern und ungültigen Eingaben vorsehen? Wie viele Reserven würden Sie einkalkulieren?

Die Ermittlung der Klassen

Die Ermittlung der wichtigsten Klassen gehört noch zur Analyse des Systems. Ihnen ist wahrscheinlich aufgefallen, dass durch die Diskussion mit dem Auftraggeber bereits einige Hauptwörter gefallen sind, die auf Klassennamen hindeuten. Da sich die Anzahl der Klassen bei dieser Problemstellung und bis zu den ersten Prototypen in Grenzen halten wird, muss die Methode der CRC-Karten nicht angewendet werden.

In diesem Fall sollen die Klassen über die Formulierung von typischen Beispielen gefunden werden. Bereits in den Aktivitätsdiagrammen wurde der Aktienbestand erwähnt, der als Ausgangspunkt der Betrachtung dienen kann.

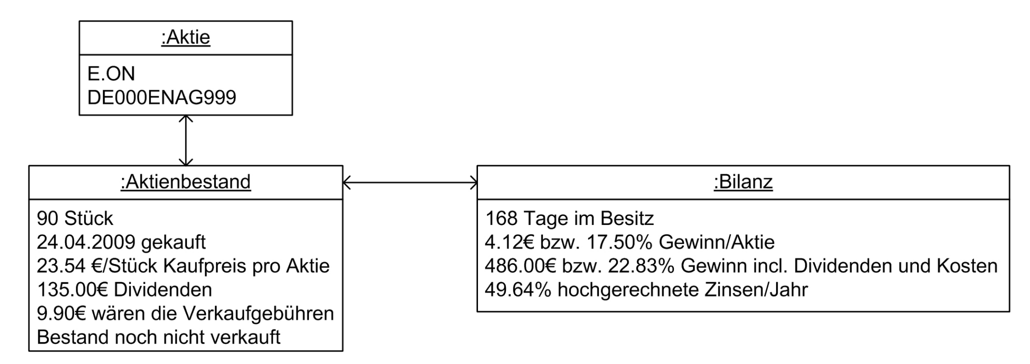


Abbildung 5.9: Objektdiagramm einer Aktie, die in einem Bestand vorliegt

Ein Aktienbestand besteht aus Aktien eines Typs. Im Beispiel aus Abbildung 5.9 wird ein Bestand der Aktie von E.ON vorgestellt. Der Bestand besteht aus 90 Aktien, die am

24.04.2009 für je 23.54 € pro Stück gekauft wurden. Beim Kauf sind 9.90 € zusätzliche Gebühren angefallen. Für diesen Bestand hat E.ON nach seiner Hauptversammlung eine Dividende von 135.00 € ausgezahlt. Der Bestand ist bislang nicht verkauft worden.

Die Bilanz des Aktienbestands nach aktuellen 168 Tagen ist ein Gewinn von 4.12 €/Aktie bzw. ein Gewinn von 17.5 %. Rechnet man die erhaltenen Dividenden hinzu und zieht die Depotgebühren für diesen Aktienbestand ab, so ergibt sich ein Gesamtgewinn von 486.00 € bzw. 22.83 %. Rechnet man die erhaltenen Zinsen linear auf ein Jahr hoch, so ergibt sich ein Zinssatz von sehr guten 49.64 %/Jahr. Es lohnt sich also, in einer Wirtschaftskrise Aktien zu kaufen.

Abbildung 5.10 zeigt den Bezug einer Aktie zu einer Kauforder. Die Anzahl, der Preis pro Aktie, das Kaufdatum und die Kaufgebühren sind mit dem Aktienobjekt verbunden. Die Aktie selbst weiß nicht unbedingt, von wem sie gekauft wurde.

Wenn der Kauf abgewickelt wurde, gehen die Daten aus dem Objekt der Kauforder in den Aktienbestand über. Das Kauforderobjekt spielt dann keine Rolle mehr in der bisherigen Modellierung.

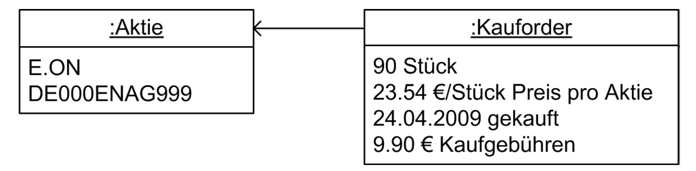


Abbildung 5.10: Objektdiagramm einer Aktie und deren Bezug zu einem Kauf

Aus den bislang ermittelten Funktionen, deren Beschreibung und den Objektdiagrammen kann ein erstes Klassendiagramm der Analysephase erstellt werden. Dieses Diagramm ist in Abbildung 5.11 abgebildet.

Ein Aktienbestand besteht aus Aktien, deren Kauf mit einer Kauforder in Zusammenhang steht. Wenn Sie Aktien verkaufen wollen, muss analog dazu eine Verkauforder existieren.

Unabhängig davon, ob ein Aktienbestand bereits verkauft wurde oder nicht, können Sie jederzeit eine Bilanz mit den Gewinnen/Verlusten aus dem Aktienbestand erstellen. Der Aktienbestand stellt nach der bisherigen Analyse eine zentrale Klasse der zu erstellenden Anwendung dar.

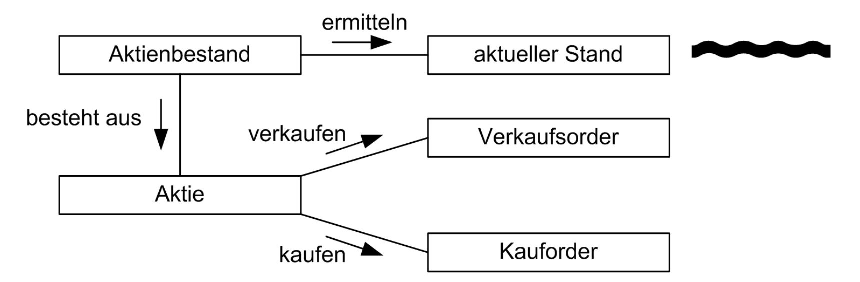


Abbildung 5.11: Erstes Klassendiagramm der Analyse für die Depotverwaltung

Der Zustand eines Aktienbestands

Bevor man sich dem Design der Anwendung widmet, kann es sinnvoll sein, von den Hauptklassen ein erstes Zustandsdiagramm anzufertigen. Der Aktienbestand kann zwei wichtige Zustände einnehmen: Entweder ist er noch im Besitz des Aktionärs oder er ist bereits verkauft.

Im ersten Fall kann der Aktionär noch Dividenden hinzufügen oder weitere Aktien nachkaufen, der Gewinn ist jedoch (bis auf die Zahlungen der Dividenden) noch nicht gesichert. Die Bilanz zeigt in diesem Fall die aktuellen Kursdaten der Aktie, wie sie an der Börse gehandelt wird.

Im zweiten Fall ist der Bestand bereits verkauft und die angezeigte Bilanz ist mitsamt ihren Gewinnen bzw. Verlusten endgültig. Vereinfachend soll zunächst nur ein gesamter Bestand verkauft werden können und nicht Teile eines Bestands. Somit führt eine Verkaufsorder wie in Abbildung 5.12 präsentiert automatisch zum Verkauf des gesamten Bestands.

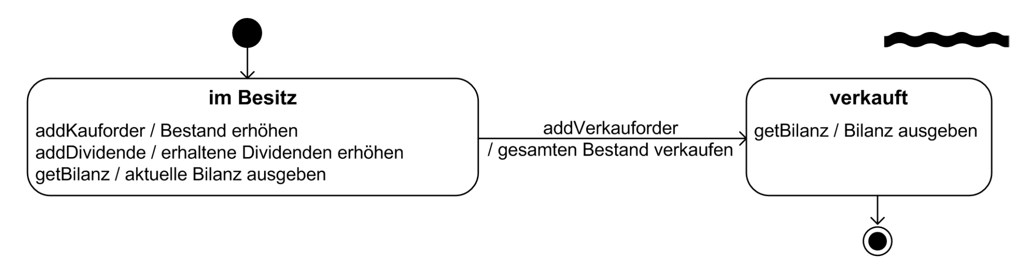


Abbildung 5.12: Zustandsdiagramm des Aktienbestands

5.1.3 Das objektorientierte Design

In der Designphase spielen fertig ausgearbeitete Klassendiagramme eine zentrale Rolle. Da PHP jedoch eine untypisierte Sprache ist, bei der die Datentypen von Eigenschaften nicht im Vorfeld definiert werden müssen und der Datentyp einer Eigenschaft sich sogar ändern kann, werden die Datentypen nicht zwingend angegeben.

Ausgangsbasis für die Weiterentwicklung des Klassendiagramms ist das Klassendiagramm der Analysephase aus Abbildung 5.11. Dort ist bereits schriftlich angegeben, dass ein Aktienbestand aus Aktien besteht. Dies entspricht einer Aggregation der UML. Da eine Aktie unter Umständen zu mehreren Beständen gehören kann und auch existiert, ohne einem Aktienbestand zugeordnet zu sein, wird von einer Komposition abgesehen. Außerdem wurde in der Analyse bereits definiert, dass ein Bestand stets aus genau einem Aktientyp besteht.

Zu einem Aktienbestand kann man auf Wunsch eine Bilanz erstellen lassen. Jede Bilanz kennt genau einen Aktienbestand.

Eine Aktie kann über eine Order gekauft oder verkauft werden. Die Order bezieht sich ebenso auf genau einen Aktientyp. Jede Order hat eine Anzahl, ein Datum, einen Einzelpreis sowie eine Gebühr. Entweder ist eine Order eine Kauforder oder eine Verkauforder. Die „Ist ein"-Beziehung wird über eine Vererbung realisiert.

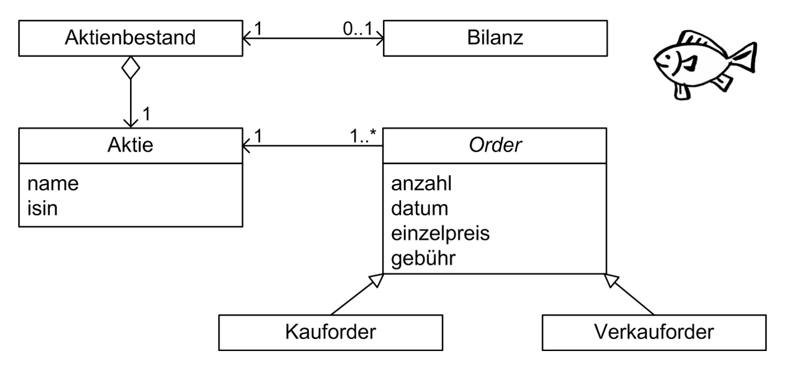


Abbildung 5.13: Ein weiterentwickeltes Klassendiagramm

Im nächsten Schritt wird in Abbildung 5.14 das Klassendiagramm so weit verfeinert, dass man aus der Zeichnung mit einem geeigneten Tool direkt die Rümpfe der Klassen erstellen kann.

Im Wesentlichen werden dabei die einzelnen privaten Eigenschaften der Klassen nun vollständig dargestellt. Die Eigenschaften resultieren aus den ermittelten Daten der objektorientierten Analyse.

Außerdem werden die wichtigsten Methoden der Klassen definiert mit deren Eingabeparametern. Typischerweise werden die Get- und Set-Methoden nicht mit angegeben, um die Komplexität der ohnehin umfangreichen Grafik zu minimieren.

Die an den Aktienbestand übergebene Kauf- bzw. Verkauforder wird nicht intern gespeichert. Stattdessen werden die Daten der Order-Objekte extrahiert und fließen in den Bestand und dessen Bilanz ein.

Sowohl eine Aktie, als auch eine Order mit deren Unterklassen verfügen lediglich über die Get- und Set-Methoden und bieten keine eigenen zusätzlichen Dienste an. Sie sind also Datencontainer für die anderen Klassen.

Neben dem Namen und der ISIN muss für jede Aktie noch ein URL angegeben werden, unter dem man den aktuellen Kurs ermitteln kann. Dieser URL soll von der Depotverwaltung eingelesen und der aktuelle Kurs mit dessen Datum ausgelesen werden.

Der Benutzer verwendet ausschließlich Dienste, die aus der Klasse des Aktienbestands stammen. Dabei handelt es sich um eine zentrale Verwaltungsklasse, die ihre Dienste für den Benutzer bereitstellt.

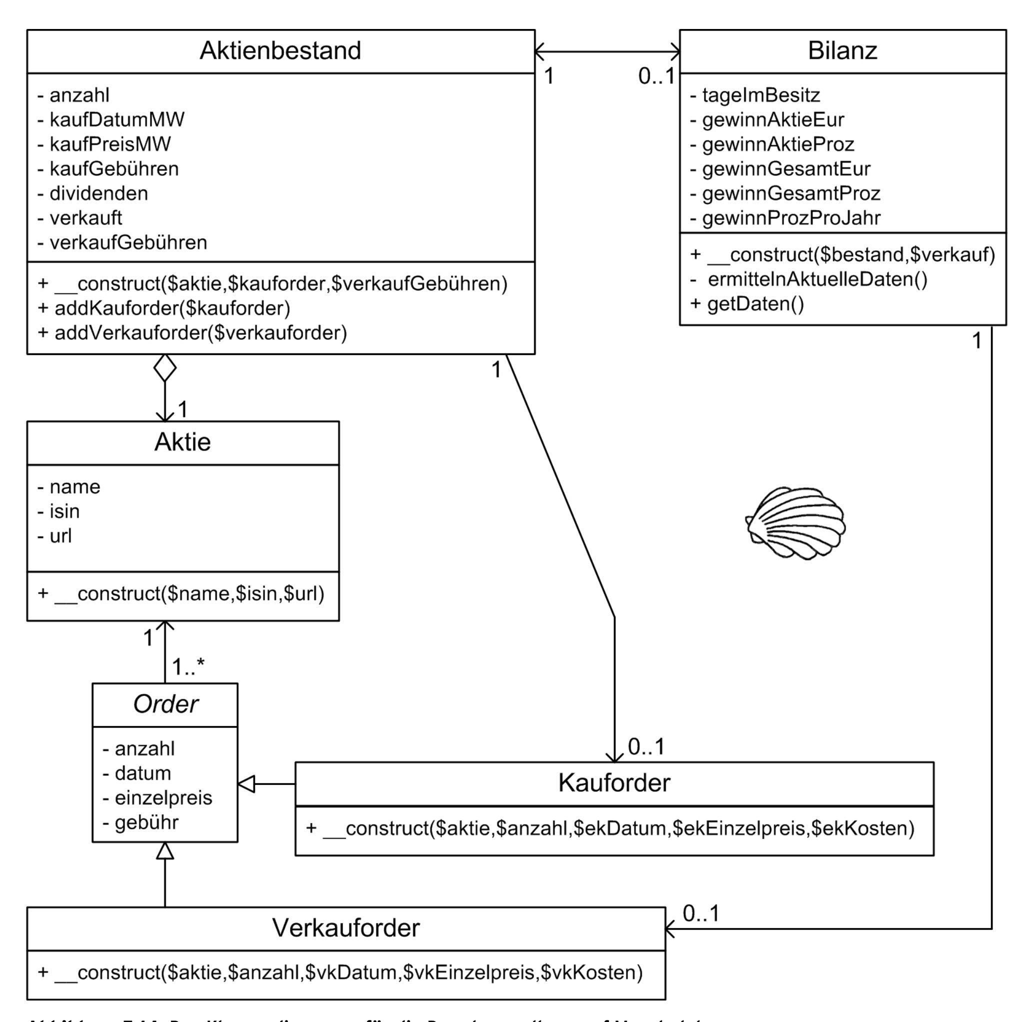


Abbildung 5.14: Das Klassendiagramm für die Depotverwaltung auf Muschelebene

Die benötigten Funktionen des Ladens und Speicherns in eine Datenbank sollen über eine leicht modifizierte Implementierung des Datenzugriffsinterfaces erfolgen, wie es bereits in Kapitel 4.2.5 dargestellt wurde. Auch hier soll wieder eine MySQL-Datenbank zum Einsatz kommen.

5.1.4 Die objektorientierte Programmierung

Bereits nach diesem Stand im objektorientierten Design empfiehlt sich der Aufbau der ersten Prototypen, um selbst ein Gefühl für die Komplexität der Aufgabenstellung zu erhalten. Ein weiterer Grund besteht darin, dass unser Auftraggeber möglichst schnell einen Lösungsansatz sehen soll, der als weitere Diskussionsgrundlage für die nächsten Schritte dient.

Hinweis

In vielen Projekten wird die OOA und OOD vollständig in einem langwierigen bürokratischen Prozess ausgearbeitet, bevor die erste Zeile Quellcode erstellt wird. Dies entspricht der Wasserfalldenkweise, bei der Fehler in der Analyse und im Design erst (zu) spät erkannt werden, was zu hohen Zusatzkosten führt. Eine agile, iterativ-inkrementelle Vorgehensweise schreibt vor, möglichst unbürokratisch schnell zur OOP vorzustoßen, nachdem die OOA und OOD in kommunikationsintensiven Workshops (unter Zuhilfenahme der UML) skizziert wurde.

Arten des Prototypings und Implementierung der Schichten

Bereits in Kapitel 3.1.3 wurden verschiedene Arten des Prototypings besprochen. Da es in unserem Fall nicht notwendig ist, dem Auftraggeber schnell „etwas zum Klicken“, also einen horizontalen GUI-Prototypen, zur Verfügung zu stellen, wurde sich für die Erstellung eines vertikalen Prototypen entschieden. Mit diesem Prototyp lässt sich insbesondere eine neue Vorgehensweise wie die Objektorientierung testen.

Abbildung 5.15 zeigt nochmals die zu erstellenden Schichten der Anwendung. Statt mit Kundendaten zu arbeiten, kommen in diesem Fall Aktien und Aktienbestände zum Einsatz. Die Benutzeroberfläche (GUI) mit ihren Schaltflächen und Eingabefeldern wird mit statischen HTML-Seiten realisiert, die zum Client und damit zum Benutzer gesendet werden. Die Antworten des Benutzers haben Einfluss auf die Fachlogik, die mit PHP 5.3 unter Anwendung der Objektorientierung umgesetzt wird. Diese Fachlogik hat wiederum Zugriff auf ein Datenzugriffsobjekt zum Laden und Speichern der Aktienbestände.

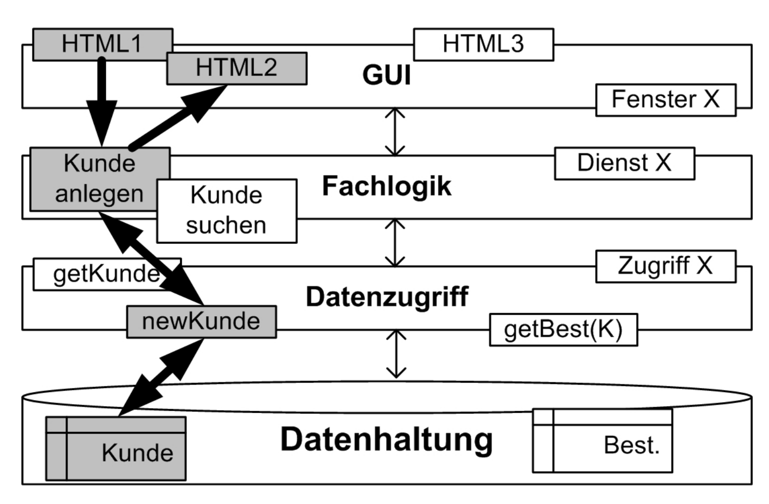


Abbildung 5.15: Beispiel eines vertikalen Prototyps

Reihenfolge und Inhalte der Iterationen

Soweit die theoretische Vorgehensweise. Wie beginnt man aber nun? Was sind die ersten Schritte? In den Kapiteln 5.1.2 und 5.1.3 wurden in Zusammenarbeit mit dem Auftraggeber bereits ein fachliches und ein technisches Modell entwickelt.

Es liegt also nahe, mit der Implementierung der Fachlogik in den ersten Prototypen zu beginnen und in einer Testmethode Objekte der Fachlogik anzulegen. Dies ist besonders dann ratsam, wenn Sie noch keine große Erfahrung mit der Objektorientierung besitzen. Änderungen im Design haben dadurch weniger Einfluss auf die gesamte Anwendung.

In der zweiten Phase werden die Parameter der Testmethode nicht fest implementiert, sondern über ein ausgefülltes HTML-Formular befüllt. Unser Auftraggeber kann dann selbst Eingaben tätigen, ein Gefühl für die Anwendung entwickeln und Feedback für die weitere Entwicklung geben.

In der letzten und dritten Phase müssen die Objekte über die Datenzugriffsschicht in der Datenbank persistent gehalten werden. Damit können Aktienbestände gespeichert und wieder geladen werden.

Im Anschluss daran müsste die nächste Iteration der Analyse und des Designs stattfinden, um die nächsten Schritte im Projektablauf zu planen. Dieser Zeitpunkt nach der dritten Phase ist auch geeignet für ein erstes Resümee der eingesetzten Ressourcen und ggf. für eine erste Teilrechnung für den Auftraggeber.

Anwenden der testgetriebenen Entwicklung

In diesem Beispielprojekt werden die ersten drei Phasen noch in dem gewöhnlichen Texteditor UltraEdit 9.00c implementiert, da die Einführung einer professionellen Entwicklungsumgebung eine individuelle Entscheidung ist. Eine Übersicht über die aktuell verbreiteten Entwicklungsumgebungen wird in Kapitel 5.3 gegeben.

Bei den ersten Schritten der Implementierung der Depotverwaltung kommt das Prinzip der testgetriebenen Entwicklung zum Einsatz. Auch hier wird noch auf hilfreiche unterstützende Werkzeuge wie PHPUnit (*http://www.phpunit.de/*) verzichtet, das auch in Kapitel 5.2.4 beschrieben wird.

An dieser Stelle soll stattdessen die Denkweise der testgetriebenen Entwicklung eingeübt werden, die eine andere Vorgehensweise der Entwickler erfordert. Die Testfälle müssen konsequent vor den zu testenden Komponenten erstellt werden.

Sie schreiben also den Quellcode zur Erzeugung von Objekten, noch bevor die entsprechenden Klassen existieren. Wie die Methoden zur Erzeugung von Objekten und die Methoden zu deren Kommunikation untereinander auszusehen haben, ist im Klassendiagramm auf Muschelebene in Abbildung 5.14 skizziert.

Damit verhindern Sie, dass der entstehende Quellcode letztlich nichts mehr mit den Gedanken und Definitionen der Analyse und des Designs zu tun hat. Sie müssen also auch in der Implementierung dienstorientiert denken, indem Sie abgeschlossene Funktionalität für die anderen Schichten zur Verfügung stellen. Die Alternative wäre eine monolithische Implementierung, deren Objekte man nachträglich nicht weiter verwenden könnte.

Die erste Phase: Implementierung der Fachlogik

Es wurde also entschieden, mit der Implementierung der Fachlogik zu beginnen und eine testgetriebene Entwicklung anzuwenden. Doch womit fängt man am besten an?

Es ist sinnvoll, mit einem Objekt zu beginnen, dessen Existenz nicht von anderen Objekten abhängt, sozusagen ein Basisobjekt. Abbildung 5.14 zeigt, dass es sich bei der Aktie um ein solches Objekt handelt. Andere Objekte können zwar Aktien kennen, die Aktie selbst kann jedoch ohne Kenntnis von anderen Objekten existieren. Eine Aktie besteht aus einem Namen und einer ISIN. Zusätzlich wurde herausgearbeitet, dass ein URL notwendig ist, um den aktuellen Kurs der Aktie abzurufen. Einen solchen URL können wir bereits der Aktie übergeben, ohne dass die Funktion zum Auslesen der aktuellen Kursdaten implementiert ist.

Der erste Schritt des Tests besteht also darin, eine Aktie erfolgreich anzulegen und deren Daten auszulesen. Im Beispiel aus Listing 5.1 wird die Aktie der Metro AG angelegt und die Daten werden über Get-Methoden ausgelesen.

```
<?php require_once("classloader.inc.php"); ?>
<html><body>
<?php
  $metro=new Aktie("METRO AG Stammaktien o.N.","DE0007257503",
                            "http://www.boerse-....ISIN=DE0007257503");
  echo 'Name: '.$metro->getName().'<br>';
  echo 'ISIN: '.$metro->getISIN().'<br>';
  echo 'URL: '.$metro->getURL().'<br>';
?>
</body></html>
```

Listing 5.1: Die erste Testklasse der Fachlogik – die Aktie

Bevor Sie nun mit der Implementierung der Aktienklasse beginnen, sollten Sie sich überlegen, welche Ausgaben Sie bei den Get-Methoden erwarten. Die Ausgabe sollte lauten:

Name: METRO AG Stammaktien o.N.

ISIN: DE0007257503

URL: http://www.boerse-....ISIN=DE0007257503

Eine andere Ausgabe würde eine fehlerhafte Implementierung bedeuten. Nun erstellen Sie die Aktienklasse, sodass diese Ausgabe erfolgreich auf dem Bildschirm erscheint. Die Klasse ist in Listing 5.2. dargestellt. Sie enthält aus Sicht eines Entwicklers keine spannenden Elemente. Die Set-Methoden sind *private* gesetzt, da sie nach der Erzeugung des Aktienobjekts nur noch ausgelesen und nicht geändert werden sollen.

```
<?php
  class Aktie{
    private $name; // Name der Aktie
```

Listing 5.2: Die Klasse der Aktie

```
    private $isin; // ISIN-Nummer der Aktie
    private $url; // URL zur Aktie, zum Abrufen des aktuelle Kurses

    public function __construct($name,$isin,$url){
      $this->setName($name); $this->setISIN($isin); $this->setURL($url);
    }

    public function getName(){
      return $this->name;
    }
    private function setName($value){
      $this->name=$value;
    }

    public function getISIN(){
      return $this->isin;
    }
    private function setISIN($value){
      $this->isin=$value;
    }

    public function getURL(){
      return $this->url;
    }
    private function setURL($value){
      $this->url=$value;
    }
  }
?>
```

Listing 5.2: Die Klasse der Aktie (Forts.)

Mit dieser Klasse wird der Test aus Listing 5.1 erfolgreich bestanden. Erkennen Sie die andere Vorgehensweise der testgetriebenen Entwicklung im Vergleich zu herkömmlicher Programmierung? Dann hätten Sie mit der Implementierung von Listing 5.2 begonnen. Das Erfüllen der zuvor definierten Tests erhöht erfahrungsgemäß auch den Spaß an der Implementierung und die Motivation der Entwickler, die bei den Werten agiler Entwicklung einen hohen Stellenwert hat.

Schon jetzt können wir mit dem zweiten Test beginnen: Aktien kann man kaufen. Dies geschieht über eine Kauforder. Eine Kauforder ist eine Order, die in unserem Klassenmodell keine zusätzlichen Methoden besitzt. Der Quellcode des zweiten Tests in Listing 5.3 legt wieder eine Aktie an, die einer Kauforder übergeben wird. Um den Erfolg des Tests zu prüfen, werden Daten der erstellten Order ausgegeben.

```
<?php require_once("classloader.inc.php"); ?>
<html><body>
<?php
  $metro=new Aktie("METRO AG Stammaktien o.N.","DE0007257503",
                            "http://www.boerse-....ISIN=DE0007257503");
  $kauforder=new Kauforder($metro,20,"06.02.2009",28.00,9.90);
  echo 'Name: '.$kauforder->getAktie()->getName().'<br>';
  echo 'Datum: '.$kauforder->getDatum().'<br>';
  echo 'Menge: '.$kauforder->getAnzahl().'<br>';
  echo 'Kurs: '.$kauforder->getEinzelpreis().'<br>';
  echo 'Gebühren: '.$kauforder->getGebühr().'<br>';
?>
</body></html>
```

Listing 5.3: Testklasse der Fachlogik – die Kauforder

Die zu erwartende Ausgabe lautet:

Name: METRO AG Stammaktien o.N.

Datum: 06.02.2009

Menge: 20

Kurs: 28

Gebühren: 9.9

Da sowohl die Kauforder als auch die Verkauforder eine Order (noch) ohne zusätzliche Funktionalität ist und alle Parameter (Anzahl, Datum, Einzelpreis, Gebühr) zu jeder Order gehören, werden die Eigenschaften in Listing 5.4 in die abstrakte Oberklasse *Order* ausgelagert. Auch hier sind die Set-Methoden wieder *private* deklariert, da eine Order (noch) nicht nachträglich änderbar sein soll.

```
<?php
  abstract class Order{
    private $aktie; private $anzahl=0;
    private $datum; private $einzelpreis; private $gebühr;

    public function __construct(
                           $aktie,$anzahl,$datum,$einzelpreis,$gebühr){
      $this->setAktie($aktie);
      $this->setAnzahl($anzahl);
      $this->setDatum($datum);
      $this->setEinzelpreis($einzelpreis);
      $this->setGebühr($gebühr);
```

Listing 5.4: Implementierung der Klasse „Order“

```
    }

    public function getAktie(){
        return $this->aktie;
    }
    private function setAktie($value){
        $this->aktie=$value;
    }

    public function getAnzahl(){
        return $this->anzahl;
    }
    private function setAnzahl($value){
        $this->anzahl=$value;
    }

    public function getDatum(){
        return $this->datum;
    }
    private function setDatum($value){
        $this->datum=$value;
    }

    public function getEinzelpreis(){
        return $this->einzelpreis;
    }
    private function setEinzelpreis($value){
        $this->einzelpreis=$value;
    }

    public function getGebühr(){
        return $this->gebühr;
    }
    private function setGebühr($value){
        $this->gebühr=$value;
    }
  }
?>
```

Listing 5.4: Implementierung der Klasse „Order" (Forts.)

Bei der Kauforder wird in Listing 5.5 lediglich der Konstruktor so verändert, dass die Eingabedaten namentlich einer Kauforder entsprechen. Aus dem Datum wird daher ein

Einkaufsdatum. Dies kann hilfreich sein, wenn später Werkzeuge eingesetzt werden sollen, die eine Hilfestellung beim Ausfüllen der Parameter geben. In diesem Fall würde die Hilfestellung *$ekDatum* anbieten.

```
<?php
  class Kauforder extends Order{
    public function __construct(
                       $aktie,$anzahl,$ekDatum,$ekEinzelpreis,$ekKosten){
      parent::__construct(
                       $aktie,$anzahl,$ekDatum,$ekEinzelpreis,$ekKosten);
    }
  }
?>
```

Listing 5.5: Implementierung der Klasse „Kauforder“

Auf die gleiche Weise wird mit einer Verkaufsorder in Listing 5.6 verfahren. Damit sind bereits 4 der 6 benötigten Klassen implementiert.

```
<?php
  class Verkauforder extends Order{
    public function __construct(
                       $aktie,$anzahl,$vkDatum,$vkEinzelpreis,$vkKosten){
      parent::__construct(
                       $aktie,$anzahl,$vkDatum,$vkEinzelpreis,$vkKosten);
    }
  }
?>
```

Listing 5.6: Implementierung der Klasse „Verkauforder“

Mit dieser Implementierung wird die gewünschte Ausgabe erzeugt. Nun gilt es, die Order in einem Aktienbestand zu speichern. Dies geschieht in Listing 5.7. Zusätzlich dazu soll eine Dividende zum Bestand hinzugefügt werden. Im zweiten Teil des Tests werden weitere Aktien gemäß den erstellten Formeln aus der Analyse (Abb. 5.8) nachgekauft und die gewichteten arithmetischen Mittelwerte für das Kaufdatum und den Kaufkurs ausgegeben.

```
<?php require_once("classloader.inc.php"); ?>
<html><body>
<?php
  $metro=new Aktie("METRO AG Stammaktien o.N.","DE0007257503",
                              "http://www.boerse-....ISIN=DE0007257503");
  $kauforder=new Kauforder($metro,200,"01.01.2010",1.00,9.90);
```

Listing 5.7: Test des Aktienbestands sowie Hinzufügen einer Dividende und zweiten Kauforder

```
  // erster Teil des Tests
  $metroBestand=new Aktienbestand($metro,$kauforder,9.9);
  $metroBestand->addDividende(135.00);
  echo 'Name: '.$metroBestand->getAktie()->getName().'<br>';
  echo 'Datum1: '.$metroBestand->getEkDatum().'<br>';
  echo 'Menge1: '.$metroBestand->getAnzahl().'<br>';
  echo 'Kurs1: '.$metroBestand->getEkEinzelpreis().'<br>';
  // zweiter Teil des Tests
  $kauforder2=new Kauforder($metro,100,"01.03.2010",2.00,9.90);
  $metroBestand->addKauforder($kauforder2);
  echo 'Name: '.$metroBestand->getAktie()->getName().'<br>';
  echo 'DatumX: '.$metroBestand->getEkDatum().'<br>';
  echo 'MengeX: '.$metroBestand->getAnzahl().'<br>';
  echo 'KursX: '.$metroBestand->getEkEinzelpreis().'<br>';
?>
</body></html>
```

Listing 5.7: Test des Aktienbestands sowie Hinzufügen einer Dividende und zweiten Kauforder (Forts.)

Die erwartete Ausgabe lautet hier:

Name: METRO AG Stammaktien o.N.

Datum1: 01.01.2010

Menge1: 200

Kurs1: 1

Name: METRO AG Stammaktien o.N.

DatumX: 21.01.2010

MengeX: 300

KursX: 1.33

In Listing 5.8 wird die gewünschte Funktionalität realisiert. Ein Aktienbestand kennt seine Aktie, die Anzahl der Aktien im Bestand sowie den Kurs und das Datum des Kaufvorgangs bzw. die errechneten Mittelwerte für den Kaufkurs bzw. das Kaufdatum. Zusätzlich werden die Gebühren des Kaufs festgehalten.

Bei mehreren Käufen kumulieren sich die Kaufgebühren. Dies gilt auf der Einnahmeseite auch für die Dividenden, die ebenfalls als Eigenschaft im Aktienbestand gespeichert werden. Auch wenn der Verkauf des Bestands im jetzigen Test noch nicht implementiert ist, wird eine Eigenschaft für die angefallenen Verlaufsgebühren ebenso vorgesehen wie ein Flag zur Abfrage, ob der Bestand verkauft ist oder nicht.

Der Konstruktor erhält die Aktie, von dem der Bestand angelegt werden soll, die erste Kauforder sowie die Verkaufgebühren, die bei einem Verkauf anfallen würden. Hier erkennen Sie bereits einen ersten Ansatz für eine Fehlerbehandlung, die prüft, ob wirk-

lich ein Aktienobjekt übergeben wird. Ist dies nicht der Fall, so wird eine Exception geworfen und der Bestand nicht angelegt. Jede einzelne Klasse muss auf diese Weise mit einer robusten Fehlerbehandlung ausgerüstet werden, indem die Eingangsparameter jeder Methode auf Gültigkeit geprüft werden.

Im Anschluss daran folgen die typischen Get- und Set-Methoden, die nicht vollständig implementiert wurden. Beachten Sie in der Methode *addDividende()*, dass die Dividendenzahlungen aufaddiert und nicht überschrieben werden. Auch hier wird eine rudimentäre Fehlerbehandlung durchgeführt.

Von besonderem Interesse ist die letzte Methode der Klasse *addKauforder($value)*, die eine weitere Kauforder in den Aktienbestand integriert. Beim ersten Kauf wird lediglich der Kaufpreis und das Kaufdatum aus der Order übernommen. Kommt ein weiterer Kauf hinzu, wird aus dem bisherigen Kaufpreis bzw. Kaufdatum und dem neuen Kaufpreis bzw. Kaufdatum der gewichtete arithmetische Mittelwert gemäß den Formeln aus der Analyse gebildet und in den Eigenschaften des Objekts unter *$kaufPreisMW* bzw. *$kaufDatumMW* abgelegt. Abschließend werden noch die Anzahl der Aktien im Bestand und die Kaufgebühren um die in der Kauforder enthaltenen Werte erhöht.

```
<?php
  class Aktienbestand{
    private $aktie;
    private $aktBestand=0; // aktuelle Anzahl an Aktien
    private $kaufDatumMW=FALSE;
    private $kaufPreisMW=0.0; // Kurs der Aktie beim Kauf
    private $kaufGebühren;
    private $dividenden=0.0; // zusätzlich erhaltene Dividenden
    private $verkaufGebühren=0.0;
    private $verkauft=FALSE; // Bestand komplett verkauft?

    public function __construct($aktie,$kauforder,$verkaufGebühren){
      if (($aktie instanceof Aktie)==FALSE){
        throw new Exception("__construct:Sie müssen ein Aktien-Objekt
                                                    übergeben!");
      }
      $this->setAktie($aktie);
      $this->addKauforder($kauforder);
      $this->setVerkaufGebühren($verkaufGebühren);
    }

    public function getAktie(){
        return $this->aktie;
    }
    private function setAktie($value){
```

Listing 5.8: Implementierung des Aktienbestands ohne Verkauf und ohne Bilanz

```
      $this->aktie=$value;
    }

    public function getVerkaufGebühren(){
      return $this->verkaufGebühren;
    }
    private function setVerkaufGebühren($value){
      if ($this->isVerkauft()){
        throw new Exception("setVerkaufGebühren:Bestand ist bereits
                                                  geschlossen!");
      }
      $this->verkaufGebühren=$value;
    }

    public function addDividende($value){
      if ($this->isVerkauft()){
        throw new Exception("addDividende:Bestand ist bereits
                                                  geschlossen!");
      }
      $this->dividenden+=$value;
    }
    public function getDividenden(){
      return $this->dividenden;
    }

    public function getEkDatum(){
      return $this->kaufDatumMW;
    }

    public function getEkEinzelpreis(){
      return $this->kaufPreisMW;
    }

    public function getKaufGebühren(){
      return $this->kaufGebühren;
    }

    public function isVerkauft(){
      return $this->verkauft;
    }
```

Listing 5.8: Implementierung des Aktienbestands ohne Verkauf und ohne Bilanz (Forts.)

```
    public function addKauforder($value){
      if ($this->isVerkauft()){
        throw new Exception("addVerkauforder:Bestand ist bereits
                                                  geschlossen!");
      }
      // arithm. gewichteter Mittelwert des EK-Preises berechnen
      if ($this->kaufPreisMW==0.0){
        // 1. Kauf
        $this->kaufPreisMW=$value->getEinzelpreis();
        $this->kaufDatumMW=$value->getDatum();
      }
      else{
        // n. Kauf
        $anz1=$this->aktBestand; $ep1=$this->kaufPreisMW;
        $d1=$this->kaufDatumMW;
        $anz2=$value->getAnzahl(); $ep2=$value->getEinzelpreis();
        $d2=$value->getDatum();
        $g=$anz2/($anz1+$anz2);
        $this->kaufPreisMW=$ep1+$ep1*$g;
        $anzTage=(strtotime($d2)-strtotime($d1))/86400;
        $Dx=date_add(new DateTime($d1),
                  new DateInterval('P'.round($g*$anzTage,0).'D'));
        $this->kaufDatumMW=$Dx->format("d.m.Y");
      }
      $this->aktBestand+=$value->getAnzahl();
      $this->kaufGebühren+=$value->getGebühr();
    }
  }
?>
```

Listing 5.8: Implementierung des Aktienbestands ohne Verkauf und ohne Bilanz (Forts.)

Im nächsten Schritt werden die Bilanzierung eines Aktienbestands und der Verkauf des gesamten Bestands implementiert. Eine Bilanz kann auch erstellt werden, wenn der Bestand noch nicht verkauft wurde. In diesem Fall sollen die Gewinne bzw. Verluste nach dem aktuellen Kurs der Aktie berechnet werden. Dieser Kurs soll von einer bestehenden Börsenhomepage ermittelt werden.

Wenn dann der Bestand verkauft wird, gibt es ein Verkaufsdatum, einen Kurs, zu dem verkauft wurde, sowie ggf. angefallene Verkaufsgebühren. Die Bilanz wird in diesem Fall aus den Verkaufsdaten ermittelt und kann im Nachhinein nicht mehr verändert werden.

In Listing 5.9 werden zwei Aktienbestände testweise verwaltet. Zunächst wird eine Commerzbank-Aktie angelegt mit ihrem Namen, der ISIN und dem URL, unter dem der aktuelle Kurs der Aktie abgefragt werden kann. Bei der Erstellung der Bilanz wird dieser URL abgefragt und die Daten aus der Homepage ausgelesen. Dann werden eine Kauforder erstellt und ein Bestand angelegt. Dieser Bestand wird dann gewinnbringend verkauft, indem eine neue Verkaufsorder über den gesamten Bestand angelegt wird. Es wurden 300 Aktien zu einem Kurs von 6.29 €/Stück eingekauft und nach ca. 6 Monaten für 8.00 €/Stück verkauft. Dabei fallen 2x9.90 € Gebühren an. Dabei ist für den Anleger interessant

- wie hoch seine Kosten beim Einkauf waren
- für welchen Preis er die Aktien letztlich verkauft hat
- wie viel Gewinn er in Euro incl. Dividenden und Kosten gemacht hat
- wie hoch sein prozentualer Gewinn incl. Dividenden und Kosten ist
- wie viel Prozent Gewinn er durchschnittlich pro Jahr erwirtschaftet hat incl. Dividenden und Kosten.

Wurde die Aktie weniger als ein Jahr angelegt, soll der bislang erhaltene Gewinn bzw. Verlust linear interpoliert werden (Abb. 5.8). Dieser Wert ist ein guter Vergleich zu einer Festgeldanlage, die üblicherweise zwischen 1.5 %/Jahr und 4.0 %/Jahr verzinst wird. Bei einer Festgeldanlage ist natürlich auch das Risiko wesentlich geringer.

Der zweite Bestand wurde von der Metro AG gekauft. Dabei wurden 100 Aktien am 01.01.2009 zu je 20.00 € erworben und es fielen zusätzlich 9.90€ Kaufgebühren an. Der aktuelle Kurs (Stand: 19.10.2009) soll aus dem Internet ermittelt und für die aktuelle Berechnung des Wertes des Aktienbestands verwendet werden.

```
<?php require_once("classloader.inc.php"); ?>
<html><body>
<?php
  $commerz=new Aktie("Commerzbank","DE0008032004",
  "http://www.boerse...ISIN=DE0008032004");
  $k=new Kauforder($commerz,300,"07.05.2009",6.29,13.15);
  $commerzBestand=new Aktienbestand($commerz,$k,9.9);
  $vk=new Verkauforder($commerz,300,"02.10.2009",8.00,9.9);
  $commerzBestand->addVerkauforder($vk);
  $data=$commerzBestand->getBilanz();
  echo 'COMMERZ<br>';
  echo 'EK: '.number_format($data[ekSumme],2).' €<br>';
  echo 'VK: '.number_format($data[aktSumme],2).' €<br>';
  echo 'Gewinn: '.number_format($data[gewinnEur],2).' € in '.
                     number_format($data[tageImBesitz],0).' Tagen<br>';
  echo 'Gewinn: '.number_format($data[gewinnProz],2).' %<br>';
```

Listing 5.9: Test des Aktienbestands mit Verkauf und Bilanz

```
  echo 'Gewinn: '.number_format($data[gewinnProzProJahr],2).
                                                          ' %/Jahr<br>';

  $metro=new Aktie("Metro AG Stammaktien o.N.","DE0007257503",
  "http://www.boerse...ISIN=DE0007257503");
  $k=new Kauforder($metro,100,"01.01.2009",20.00,9.90);
  $metroBestand=new Aktienbestand($metro,$k,9.9);
  $metroBestand->addDividende(135.00);
  $data=$metroBestand->getBilanz();
  echo 'METRO<br>';
  echo 'EK: '.number_format($data[ekSumme],2).' €<br>';
  echo 'VK: '.number_format($data[aktSumme],2).' €<br>';
  echo 'Gewinn: '.number_format($data[gewinnEur],2).' € in '.
                     number_format($data[tageImBesitz],0).' Tagen<br>';
  echo 'Gewinn: '.number_format($data[gewinnProz],2).' %<br>';
  echo 'Gewinn: '.number_format($data[gewinnProzProJahr],2).
                                                          ' %/Jahr<br>';
?>
</body></html>
```

Listing 5.9: Test des Aktienbestands mit Verkauf und Bilanz (Forts.)

Die erwartete Ausgabe sieht durch die unmittelbare Erholung nach der Wirtschaftskrise sehr vielversprechend aus:

COMMERZ

EK: 1,900.15 €

VK: 2,390.10 €

Gewinn: 489.95 € in 148 Tagen

Gewinn: 25.78 %

Gewinn: 63.63 %/Jahr

METRO

EK: 2,009.90 €

VK: 3,945.00 €

Gewinn: 2,070.10 € in 291 Tagen

Gewinn: 103.00 %

Gewinn: 129.29 %/Jahr

Die existierende Klasse des Aktienbestands wird dabei um zwei Methoden erweitert. Einerseits muss die Verkaufsorder in den Bestand mit aufgenommen werden. Dies

geschieht über die Methode *addVerkauforder($value)*, die eine Verkaufsorder als Eingabeparameter übergeben bekommt. Dabei erfolgt zunächst eine Prüfung, ob der Bestand bereits verkauft ist. Nur wenn dies nicht der Fall ist, kann der Verkauf eingerechnet werden. Dann wird überprüft, ob die Verkaufsorder den gesamten Bestand zum Verkauf enthält. Dies ergibt in den ersten Prototypen eine vereinfachte Berechnung. Streng genommen sollte auch überprüft werden, ob es sich bei *$value* wirklich um ein Verkaufsorderobjekt handelt. Darauf wird an dieser Stelle aber zu Zwecken der Übersichtlichkeit verzichtet. Sind alle Prüfungen bestanden, wird der Bestand als *verkauft* markiert und ein neues Bilanzobjekt angelegt. Dieses Objekt bekommt den Aktienbestand selbst sowie die Verkaufsorder zur Auswertung übergeben.

Die zweite Erweiterung der Aktienbestandklasse beinhaltet die Rückgabe der Bilanz in der Methode *getBilanz()*. Wenn der Bestand noch nicht verkauft wurde, wird eine neue Bilanz auf Basis der aktuellen Kursdaten erstellt. Statt der Verkaufsorder aus dem ersten Fall werden lediglich die Verkaufsgebühren an den Konstruktor des Bilanzobjekts übergeben, die bei einem Verkauf anfallen würden.

Wenn der Bestand bereits verkauft wurde, wird das schon existierende Bilanzobjekt verwendet. In beiden Fällen werden die Daten der Bilanz als Rückgabe der Methode an den Aufrufer übergeben.

```
<?php
  class Aktienbestand{
    ...

    // Verkaufen
    public function addVerkauforder($value){
      if ($this->isVerkauft()){
        throw new Exception("addVerkauforder:Bestand ist bereits
                                                  geschlossen!");
      }
      // Vereinfachung:
      if ($value->getAnzahl()!=$this->getAnzahl()){
        throw new Exception("addVerkauforder:Kann nur den gesamten
                                            Bestand verkaufen!");
      }
      $this->verkauft=TRUE;
      // Bilanz der Trading-Aktion -> Bilanz-Objekt
      $this->bilanz=new Bilanz($this,$value);
    }

    public function getBilanz(){
      if ($this->isVerkauft==FALSE){ // Bestand noch nicht verkauft...
        // -> Bilanz hängt vom aktuellen Kurs ab
```

Listing 5.10: Erweiterte Klasse des Aktienbestands

```
        $bilanz=new Bilanz($this,$this->verkaufGebühren);
        return $bilanz->getDaten();
      }
      // Bilanz ist ggf. schon vorhanden aus dem Verkauf und
      // konstant, weil der Bestand ja bereits verkauft wurde
      return $this->bilanz->getDaten();
    }
  }
?>
```

Listing 5.10: Erweiterte Klasse des Aktienbestands (Forts.)

Das Problem verlagert sich also vom Aktienbestand in die Erstellung der Bilanz, in der das Auslesen der aktuellen Kurse von der Homepage sowie alle mathematischen Berechnungen vorgenommen werden. Listing 5.11 beschreibt die Bilanzklasse, die neben dem Konstruktor aus den Methoden *ermittelnAktuelleDaten()* zum Auslesen der Homepagedaten und *getDaten()*, die ein Datenfeld mit den Ergebnissen der Bilanzberechnung zurückgibt.

Der Inhalt der beiden Methoden wird im Anschluss an den Quellcode genauer beschrieben.

Der Konstruktor der Klasse erhält eine Referenz auf den Aktienbestand sowie die Verkaufsorder bzw. als Alternative dazu die theoretischen Verkaufsgebühren als Eingabeparameter übergeben, falls der Bestand noch nicht verkauft wurde. Beide Eingabeparameter werden als Eigenschaften des Objekts gespeichert.

```
<?php
  class Bilanz{
    private $bestand=NULL;
    private $verkauf=NULL;

    public function __construct($bestand,$verkauf){
      $this->bestand=$bestand;
      $this->verkauf=$verkauf;
    }

    private function ermittelnAktuelleDaten(){
      $ret=array();
      // Homepage auslesen
      $url=$this->bestand->getAktie()->getURL();
      if ($url==""){
        $ret[Kurs]=FALSE; $ret[Datum]=FALSE;
        $ret[Zeit]=FALSE; return $ret;
```

Listing 5.11: Die neue Klasse „Bilanz“

```
    }
    $handle = fopen ($url, "r");
    $data="";
    while (!feof($handle)) $data .= fgets($handle, 4096);
    fclose ($handle);
    // Kurs auslesen
    $search='<td class="column-datavalue2 last strong">';
    $searchE='</td>';
    $pos1=strpos($data,$search,1);
    if ($pos1>0){
      $pos1E=strpos($data,$searchE,$pos1+strlen($search));
      $AktKurs=(str_replace(",",".",substr($data,$pos1+strlen($search),
                                        $pos1E-$pos1-strlen($search))));
    }
    else{
      $ret[Kurs]=FALSE;
    }
    // Datum und Zeit auslesen
    $search='<td class="column-datavalue2 last">';
    $pos2=0;
    $pos2=strpos($data,$search,$pos1+1);
    if ($pos2>0){
      $ret[Datum]=substr($data,$pos2+strlen($search),10);
      $ret[Zeit]=substr($data,$pos2+strlen($search)+11,5);
    }
    else{
      $ret[Datum]=FALSE;
      $ret[Zeit]=FALSE;
    }
    return $ret;
  }

  public function getDaten(){
    $data=array();
    $data[aktienname]=$this->bestand->getAktie()->getName();
    $data[aktienISIN]=$this->bestand->getAktie()->getISIN();
    $data[aktienURL]=$this->bestand->getAktie()->getURL();
    $data[anzahl]=$this->bestand->getAnzahl();
    $data[dividenden]=$this->bestand->getDividenden();
    $data[ekDatum]=$this->bestand->getEkDatum();
```

Listing 5.11: Die neue Klasse „Bilanz“ (Forts.)

```
      $data[ekGebühr]=$this->bestand->getKaufGebühren();
      $data[ekEinzelpreis]=$this->bestand->getEkEinzelpreis();
      if ($this->bestand->isVerkauft()){
        // alles wurde bereits verkauft
        $data[aktDatum]=$this->verkauf->getDatum();
        $data[aktGebühr]=$this->verkauf->getGebühr();
        $data[aktEinzelpreis]=$this->verkauf->getEinzelpreis();
      }
      else{
        // $verkauf wären die aktuellen Verkaufsgebühren
        $akt=$this->ermittelnAktuelleDaten();
        $data[aktDatum]=$akt[Datum];
        $data[aktGebühr]=$verkauf;
        $data[aktZeit]=$akt[Zeit];
        $data[aktEinzelpreis]=$akt[Kurs];
      }
      $data[tageImBesitz]=(strtotime($data[aktDatum])-
                                 strtotime($data[ekDatum]))/86400;
      $data[ekSumme]=$data[ekEinzelpreis]*$data[anzahl]+$data[ekGebühr];
      $data[aktSumme]=$data[aktEinzelpreis]*$data[anzahl]-
                                                   $data[aktGebühr];
      $data[diffAktieEur]=$data[aktEinzelpreis]-$data[ekEinzelpreis];
      $data[diffAktieProz]=($data[aktEinzelpreis]/$data[ekEinzelpreis]-
                                              1)*100; // ohne Kosten
      $data[gewinnEur]=$data[aktSumme]-$data[ekSumme]+$data[dividenden];
      $data[gewinnProz]=$data[gewinnEur]/$data[ekSumme]*100;
      $data[gewinnProzProJahr]=100*($data[gewinnEur]/$data[tageImBesitz]
                                            *365.25)/$data[ekSumme];
      if ($data[gewinnProzProJahr]<0) $data[gewinnProzProJahr]=0;
      return $data;
    }
  }
?>
```

Listing 5.11: Die neue Klasse „Bilanz" (Forts.)

Im ersten Schritt wird die Methode *ermittelnAktuelleDaten()* beschrieben. Die Homepage bietet den kostenlosen Dienst an, stets aktuelle Börsenkurse bereitzustellen. Abbildung 5.16 zeigt die Darstellung der Daten in der HTML-Ausgabe. Interessant ist dabei neben dem Preis auch das Datum und die Uhrzeit der Ermittlung des Kurses. Es sollen die Daten des Handelsplatzes Frankfurt verwendet werden, da unser Auftraggeber der RAUB-Bank ebenso den Handel an der Frankfurter Börse für seine Kunden anbietet.

Kursinformationen		
	Xetra	Frankfurt
Letzter Preis	8,420	8,45
Datum, Zeit	19.10.2009 17:35	19.10.2009 19:53

Abbildung 5.16: Aktuelle Kursdaten der HTML-Datei

Der entsprechende HTML-Quellcode kann im Internetbrowser des Clients ausgelesen werden. Er ist im Folgenden dargestellt, wobei einige zusätzliche Zeilenumbrüche zur Übersichtlichkeit eingefügt wurden:

<th class="column-datacaption first"></th>

<th class="column-datavalue1 strong">Xetra</th><th class="column-datavalue2 strong last">Frankfurt</th></tr><tr>

<td class="column-datacaption first strong">Letzter Preis</td><td class="column-datavalue1 strong">8,420</td><td class="column-datavalue2 last strong">8,45</td>

</tr><tr class="odd"><td class="column-datacaption first">Datum, Zeit</td>

<td class="column-datavalue1">19.10.2009 17:35</td>

<td class="column-datavalue2 last">19.10.2009 19:53</td>

</tr><tr>

Ähnlich wie bei einem Dateizugriff wird mit dem Befehl *$handle = fopen (...)* unter Angabe des URL eine Referenz auf die betreffende Homepage erstellt. Vorher wird jedoch geprüft, ob der URL-String nicht leer ist. Eine leere oder auch ungültige Zeichenkette sorgt dafür, dass *fopen* blockiert und in einen Timeout läuft, der die Rückgabe der Funktion stark verzögert.

Die anschließende *while*-Schleife liest den HTML-Text der Seite so lange ein, bis das Ende erreicht ist. Dabei werden mit dem Befehl *$data .= fgets($handle, 4096);* jeweils 4kByte-Blöcke eingelesen und der HTML-Quellcode als Zeichenkette in der Variablen *$data* gespeichert. Mit dem Befehl *fclose ($handle);* wird die Verbindung zur Internetseite dann wieder geschlossen, deren Inhalt Sie nun als Zeichenkette vorliegen haben.

Wie jede andere Zeichenkette, können Sie *$data* jetzt mit den mächtigen Zeichenkettenfunktionen von PHP durchsuchen. Die erste Suche konzentriert sich auf die Zeichenfolge *<td class="column-datavalue2 last strong">*. Die Position, an der die Zeichenfolge gefunden wurde, wird unter *$pos1* gespeichert. Die Zeichenfolge kommt nur einmalig im HTML-Code vor. Die Position ist nicht größer als 0, wenn die Zeichenfolge nicht gefunden wurde. In diesem Fall wird der ermittelte Kurswert auf *$ret[Kurs]=FALSE;* gesetzt.

Im Anschluss daran wird das nächste *</td>* hinter der gefundenen Position gesucht. Dazwischen steht dann der Kurs, der extrahiert und in *$AktKurs* gespeichert wird. Zusätzlich wird das Kommatrennzeichen über die Funktion *str_replace* in einen Punkt umgewandelt, sodass *$AktKurs* als Gleitkommazahl gesehen wird. Auf diese Weise kann man mit dem extrahierten Wert rechnen.

Nach dem gleichen Prinzip werden das Datum und die Uhrzeit ausgelesen, zu der der Kurs ermittelt wurde. Hier wird nach der einmalig im HTML-Code vorkommenden Zei-

chenkette *<td class="column-datavalue2 last">* gesucht. Da das Datum mit *dd.mm.yyyy hh:mm* ein festes Format und im Gegensatz zu einem Aktienkurs (1,23 € oder 123,45 €) eine feste Anzahl an Stellen hat, werden die nächsten 16 Stellen hinter der gesuchten Zeichenkette extrahiert. Die ersten 10 Stellen werden als Datum in der Variablen *$ret[Datum]* und die Stellen 11 bis 15 als Uhrzeit in der Variablen *$ret[Zeit]* abgelegt. Wenn dies nicht erfolgreich ist, werden diese beiden Variablen ebenfalls mit *FALSE* belegt. Zur weiteren Verarbeitung wird das Datenfeld mit dem Kurs, dem Datum und der Zeit dann als Rückgabewert der Funktion zurückgegeben.

Die zweite Methode der Bilanz, *getDaten()*, gibt die Ergebnisse der Bilanzberechnung als Datenfeld zurück. Aus dem Aktienbestand werden zunächst ausgelesen

- der Name der Aktie
- die ISIN der Aktie
- der URL der Aktie

Im Anschluss daran werden die Daten des Einkaufs in das Datenfeld hinzugefügt. Dazu gehören

- die Anzahl der Aktien im Bestand
- die ggf. erhaltenen Dividendenzahlungen
- das Datum des Einkaufs (bzw. der arithmetisch gewichtete Mittelwert bei mehreren Einkäufen)
- die Gebühren beim Einkauf
- der Einzelpreis der eingekauften Aktien (bzw. der arithmetisch gewichtete Mittelwert bei mehreren Einkäufen).

Der nächste Teil der Bilanz ist abhängig davon, ob der Bestand bereits verkauft wurde oder nicht. Wenn er bereits verkauft wurde, so besitzt das Bilanzobjekt eine Verkaufsorder. Daraus werden extrahiert und dem Datenfeld hinzugefügt:

- das Datum des Verkaufs
- die Verkaufsgebühren
- der Einzelpreis der Aktie, zu dem verkauft wurde

Wurde der Bestand noch nicht verkauft, werden zunächst die aktuellen Daten von der Homepage ausgelesen. Aus diesen Daten werden ausgegeben:

- das aktuelle Datum
- die Zeit, zu dem der Kurs ermittelt wurde
- der aktuelle Einzelpreis, also der Kurs selbst

Zusätzlich dazu werden die Verkaufsgebühren aus dem Konstruktor der Bilanz ausgegeben, die bei einem Verkauf anfallen würden.

Im Anschluss daran werden die eigentlichen Berechnungen der Bilanz durchgeführt. Dabei werden die folgenden Fakten ermittelt und in das Ausgabefeld *$data* hinzugefügt, die auch später in der Ausgabe erscheinen sollen:

- *tageImBesitz* beschreibt die Anzahl der Tage, die der Bestand schon existiert. Dazu wird das Datum des Einkaufs und das aktuelle Datum (entweder das jetzige Datum, falls der Bestand noch nicht verkauft wurde oder das Datum des Verkaufs) zunächst in Sekunden umgewandelt mit der Funktion *strtotime*. Die erhaltene Differenz der beiden Werte ergibt dann die Anzahl der Sekunden, die der Aktienbestand im Besitz des Anlegers war bzw. ist. Dividiert man den Wert durch 86 400, so erhält man die Anzahl der Tage, die die Aktie im Besitz war bzw. ist, da ein Tag 24 Stunden hat, eine Stunde 60 Minuten und eine Minute 60 Sekunden hat (24x60x60=86 400).
- *ekSumme* beschreibt den Betrag, die der Anleger für den Aktienbestand bezahlen musste. Sie besteht aus dem (durchschnittlichen) Kurs der Aktie beim Einkauf, multipliziert mit der Anzahl der gekauften Aktien. Hinzu kommen noch die angefallenen Kaufgebühren.
- *aktSumme* beschreibt den Wert des Bestands. Dies ist der aktuelle Kurs bzw. der Verkaufskurs multipliziert mit der Anzahl. Davon müssen die Verkaufsgebühren noch abgezogen werden.
- *diffAktieEur* ist die Differenz des Kurses beim Einkauf zu dem aktuellen Kurs bzw. zum Kurs, zu dem verkauft wurde.
- *diffAktieProz* ist dieselbe Differenz, jedoch in Prozent. Bei beiden Differenzen wird nur der Kurs betrachtet und keine angefallenen Kosten und Dividendenzahlungen.
- *gewinnEur* ist der Gewinn (oder Verlust) in Euro, die der Aktienbestand abgeworfen hat. Dazu wird der Wert beim Einkauf vom aktuellen Wert des Bestands bzw. der Wert beim Verkauf abgezogen. Da hier die Kosten bereits berücksichtigt sind, müssen nur noch die erhalteten Dividenden hinzugefügt werden.
- *gewinnProz* ist dieser Gewinn, umgerechnet in Prozent.
- *gewinnProzProJahr* rechnet den ermittelten Gewinn auf ein Jahr hoch. Dazu wird der erhaltene Gewinn durch die Anzahl der Tage geteilt, die sich der Aktienbestand im Besitz des Anlegers befindet. Dann haben Sie den durchschnittlichen Gewinn pro Tag ermittelt. Dieser durchschnittliche Gewinn pro Tag wird mit den Tagen eines Jahres, also mit 365.25, wegen des Schaltjahres alle 4 Jahre, multipliziert. Dieser Jahresgewinn wird dann noch in Prozent umgerechnet. Dieser Wert dient zum Vergleich mit einer Festgeldanlage.

Wenn Sie mit dem Aktienbestand Verlust gemacht haben, wäre *gewinnProzProJahr* negativ. Da Sie jedoch selbst im Falle der Insolvenz der Aktiengesellschaft kein Geld draufzahlen müssen, wird der prozentuale Gewinn pro Jahr bei einem negativen Wert zum Abschluss der Methode *getDaten()* auf 0 gesetzt.

Die Erweiterung der Klasse Aktienbestand sowie die Klasse Bilanz sorgen dafür, dass die erwartete Ausgabe des Tests erfüllt wird. Die einzelnen direkten Tests mit kleinen PHP-Skripten sind damit abgeschlossen. Im nächsten Schritt wird nun die erste Benutzeroberfläche für unseren Auftraggeber erstellt.

Die zweite Phase: Die Schnittstelle zum Benutzer

Für den Prototyp der Benutzeroberfläche (GUI) wird eine Menüführung benötigt, durch die unser Auftraggeber die von ihm definierten Anwendungsfälle (Abb. 5.2) wieder findet. Dadurch wird eine geschäftsprozessnahe Umsetzung der Anforderungen unterstützt.

Sowohl die Verwendung von JavaScript-Menüs als auch HTML-Frames sind veraltet. Sie erlauben kein direktes Bookmarking und sind auch hinderlich für einen barrierefreien Zugriff. Die gesamte Darstellung soll daher in einer einzigen HTML-Datei möglichst ohne Verwendung von aktiven Skripten beim Client erfolgen.

Gleichzeitig sind Erweiterungen und Änderungen in der Menüführung sehr wahrscheinlich. Schon ab ca. 50 Bildschirmmasken ist eine Änderung vom Entwickler oder Designer nur nervenaufreibend konsistent durchzusetzen. Die Bildschirmmasken werden also nicht vollständig auf dem Webserver gespeichert, sondern modular zusammengesetzt. Für die kleine Beispielanwendung der Depotverwaltung genügt die in Abbildung 5.17 beschriebene Aufteilung.

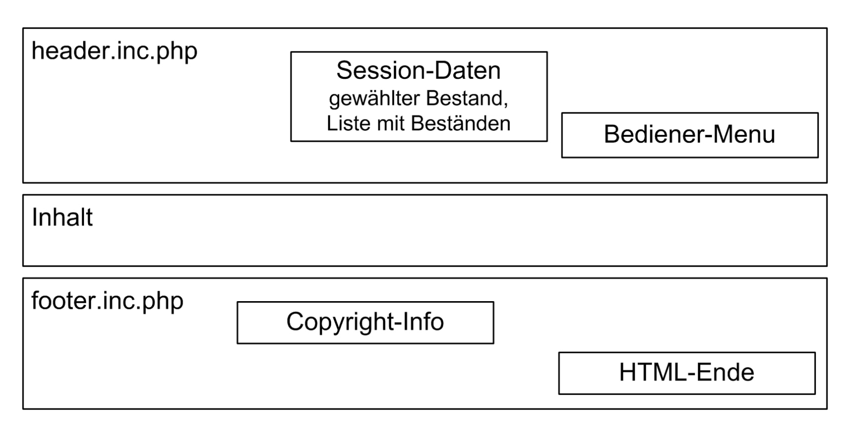


Abbildung 5.17: Aufteilung der Bedienführung

In der *header.inc.php* beginnt der HTML-Code. Dort soll auch die PHP-Session des Benutzers initialisiert werden. Die Menüführung soll auch einmalig in der Header-Datei untergebracht werden.

Der zweite Teil der HTML-Ausgabe besteht aus dem eigentlichen Inhalt, den der Benutzer ausgewählt hat. Möchte er beispielsweise einen neuen Aktienbestand anlegen, so sind dazu einige Eingabeparameter nötig. Diese Parameter sollen im Inhaltsteil in einem HTML-Formular eingegeben werden. Genau dies ist die Aufgabe der *View* im MVC-Paradigma (Abb. 3.28). Die Weiterleitung des ausgefüllten HTML-Formulars erfolgt dann an eine PHP-Datei, die den Controller des MVC darstellt. Dort werden die eingegebenen Daten ggf. auf Gültigkeit geprüft und an das Datenmodell übergeben. Der Erfolg dieser Aktion wird dann wiederum an den Benutzer weitergegeben.

Die *footer.inc.php* enthält dann lediglich einen einheitlichen Abschluss für die HTML-Darstellung, wie eine Copyright-Information und das eigentliche Ende der HTML-Datei.

Die Realisierung des Konzepts erfolgt dadurch, dass jede PHP-Inhaltsdatei den Header und den Footer inkludiert und diese beiden Dateien nur einmalig vorhanden sind. Dies ist der erste Ansatz für eine zentrale Verwaltung des Layouts und der Menüführung. Bei größeren Projekten wird ein solches Konzept weiter ausgebaut, indem die HTML-Darstellung von einem Objekt einer Template-Klasse übernommen wird. Das Darstellungs-Template enthält dann unter anderem Eigenschaften für die Farbdarstellung und den gesamten Stil der Präsentation. Dies kann beispielsweise über die Parametrierung des Objekts mit CSS-Dateien erfolgen.

Damit unser Auftraggeber die bislang nur exemplarisch getestete Funktionalität der PHP-Anwendung prüfen kann, wird in Abbildung 5.18 ein typisches Szenario in einem Aktivitätsdiagramm auf Meeresspiegelebene (also auf der Ebene des Benutzers) zusammengestellt, welches einen Großteil der geforderten Anwendungsfälle abdeckt. Dieses Szenario wird im Folgenden über das zu erstellende GUI abgedeckt.

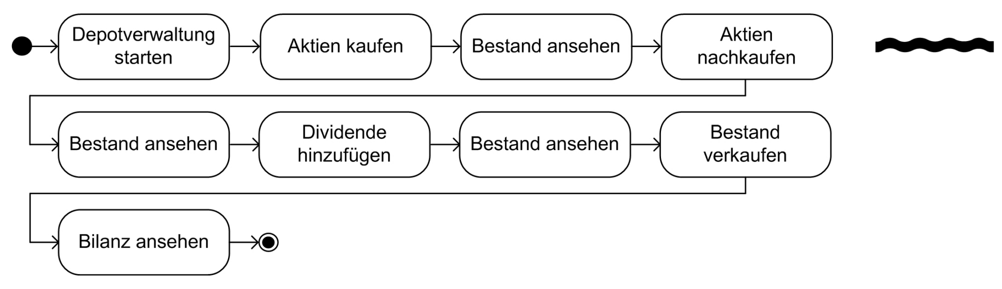


Abbildung 5.18: Szenario für die Schnittstelle zum Benutzer

Der erste Schritt besteht also darin, die Depotverwaltung zu starten. Die entsprechende Inhaltsdatei soll eine leicht in HTML zu realisierende Willkommensmeldung darstellen und sowie das Menü der Depotverwaltung präsentieren. Wie in jede Inhaltsdatei werden der Header und der Footer eingebunden. Der Inhalt dieser beiden Rahmendateien wird in den Listings 5.12 und 5.13 dargestellt.

```
<?php
  session_start();
  require_once("classloader.inc.php");
  if (!isset($_SESSION[Bestand])) $_SESSION[Bestand]=NULL;
  if (!isset($_SESSION[BestandAnzahl])) $_SESSION[BestandAnzahl]=0;
  if (isset($_GET[w])){
    $_SESSION[BestandGewählt]=$_GET[w];
  }
  else{
    if (!isset($_SESSION[BestandGewählt])) $_SESSION[BestandGewählt]=-1;
  }
?>
<html><body>
```

Listing 5.12: header.inc.php

```
<center>
<h1>Depotverwaltungsprogramm</h1>
<h3>Gewählter Bestand: 
<?php
  if ($_SESSION[BestandGewählt]==-1){
    $status="disabled";
    echo '---';
  }
  else{
    $status="";
    $bestandGewählt=unserialize($_SESSION[Bestand]
                                           [$_SESSION[BestandGewählt]]);
    echo $bestandGewählt->getAktie()->getName();
  }
?>
</h3>
<table border="0">
<tr>
<td><form action="laden.php" method="post">
  <input type="submit" value="     laden     "/>
</form></td>
<td><form action="speichern.php" method="post">
  <input type="submit" value="   speichern   " <?php echo $status?>/>
</form></td>
<td><form action="uebersicht.php" method="post">
  <input type="submit" value="Übersicht / wählen" <?php echo $status?>/>
</form></td>
<td><form action="neu.php" method="post">
  <input type="submit" value="         neu        "/>
</form></td>
<td><form action="hinzukaufen.php" method="post">
  <input type="submit" value="  hinzukaufen  " <?php echo $status?>/>
</form></td>
<td><form action="dividende.php" method="post">
  <input type="submit" value="neue Dividende" <?php echo $status?>/>
</form></td>
<td><form action="verkaufen.php" method="post">
  <input type="submit" value="   verkaufen   " <?php echo $status?>/>
</form></td>
</tr>
```

Listing 5.12: header.inc.php (Forts.)

```
</table>
<hr>
</center>
```

Listing 5.12: header.inc.php (Forts.)

In der PHP-Session (Kap. 2.2) wird vor allem die Liste der Aktienbestandsobjekte in der Referenz *$_SESSION[Bestand]* verwaltet. Zusätzlich wird festgehalten, wie viele Aktienbestände im Depot liegen (*$_SESSION[BestandAnzahl]*) und welcher Aktienbestand ggf. ausgewählt wurde (*$_SESSION[BestandGewählt]*). Der gewählte Bestand kann über eine HTTP-Get Anfrage (*$_GET[w]*) an jede Datei übergeben werden, die die *header.inc.php* einbindet.

Im Anschluss daran beginnt die Ausgabe des HTML-Codes. Sie ist abhängig davon, ob bereits ein Aktienbestand gewählt wurde oder nicht. Ist dies nicht der Fall, wird zunächst eine Statusvariable *$status* mit dem Textwert *disabled* belegt, die später im Menü der HTML-Datei noch Verwendung findet. Wurde bereits ein Bestand gewählt, so bleibt die Statusvariable leer. Im zweiten Schritt wird das gewählte Aktienbestandsobjekt aus der Liste der Bestände in die Variable *$bestandGewählt* extrahiert. Da die Liste der Objekte in einer Session nicht direkt zugegriffen werden kann, muss das entsprechende Objekt zunächst deserialisiert werden (Kap. 4.1.2). Im Anschluss daran können Sie aus der Aktie des Bestands den Namen auslesen und ausgeben.

Hinweis

Erkennen Sie, wie sich die objektorientierte Denkweise der Aktienbestände mit der prozeduralen Denkweise bei der Darstellung des HTML-Codes vereinigt?

Im Anschluss daran folgt im Header eine HTML-Tabelle mit einer Zeile und sieben Spalten, in der sich jeweils HTML-Formulare befinden, die auf die vom Auftraggeber definierten Anwendungsfälle verweisen. Sie bieten die Dienste an, um

- Aktienbestände aus der Datenbank zu laden
- Bestände wieder in die Datenbank abzuspeichern
- eine Übersicht über alle Bestände anzuzeigen und ggf. einen Bestand auszuwählen
- einen neuen Bestand anzulegen
- Aktien zu dem gewählten Bestand hinzuzukaufen
- eine Dividende zu dem gewählten Bestand hinzuzufügen
- den gewählten Bestand zu verkaufen

Wenn genau ein Bestand in der Depotverwaltung existiert, soll er stets angewählt sein. Wenn mehr als ein Aktienbestand existiert, muss stets genau ein Bestand angewählt sein. Zu Beginn der Anwendung existiert jedoch noch kein Bestand in der Depotverwaltung. Deshalb sind nur zwei Anwendungsfälle ausführbar, nämlich

- Aktienbestände aus der Datenbank laden
- einen neuen Bestand anlegen

Alle anderen Anwendungsfälle können deaktiviert werden, indem die entsprechenden Schaltflächen im HTML-Formular über den Text *disabled* grau dargestellt und inaktiv gesetzt werden. Genau dies geschieht über die Ausgabe der Variablen *$status*, die entweder den Wert *disabled* enthält oder leer ist.

Die Footer-Datei enthält lediglich einen textuellen Copyright-Hinweis und das Ende der HTML-Datei.

```
<center>
<hr>
Copyright by Dr. Frank Dopatka
</center>
</body></html>
```

Listing 5.13: footer.inc.php

Die Willkommensseite ist die erste und einfachste Inhaltsseite. Sie inkludiert den Header und den Footer, die nun die Session verwalten, das Menü und den HTML-Rumpf.

```
<?php require_once("header.inc.php"); ?>
<center>
<h1>Herzlich Willkommen!</h1>
</center>
<?php require_once("footer.inc.php"); ?>
```

Listing 5.14: start.php (View)

Wenn Sie nun die *start.php* ausführen, sehen Sie die in Abbildung 5.19 dargestellte Startseite.



Abbildung 5.19: Startmaske der Depotverwaltung

Der nächste Schritt im Szenario besteht darin, einen neuen Aktienbestand anzulegen. Dazu klicken Sie auf die Schaltfläche *neu*, die zu der Datei *neu.php* verweist. Diese Datei ist in Listing 5.15 dargestellt.

```
<?php require_once("header.inc.php"); ?>
<form action="neu2.php" method="post">
<table border="0" width="100%">
<tr><td align="right" width="50%">
  Name der Aktie:
</td><td align="left" width="50%">
  <input type="text" name="name" size="30"/>
</td></tr>
<tr><td align="right" width="50%">
  ISIN der Aktie:
</td><td align="left" width="50%">
  <input type="text" name="isin" size="30"/>
</td></tr>
<tr><td align="right" width="50%">
  URL der Aktiendaten:
</td><td align="left" width="50%">
  <input type="text" name="url" size="30"/>
</td></tr>
<tr><td align="right" width="50%">
  Anzahl gekaufter Aktien:
</td><td align="left" width="50%">
  <input type="text" name="anz" size="5"/> Stück
</td></tr>
<tr><td align="right" width="50%">
  Kurs beim Kauf:
</td><td align="left" width="50%">
  <input type="text" name="kaufkurs" size="10"/> ?
</td></tr>
<tr><td align="right" width="50%">
  Kauf-Datum:
</td><td align="left" width="50%">
  <input type="text" name="kaufdatum" size="10"/>
</td></tr>
<tr><td align="right" width="50%">
  Kauf-Gebühren:
</td><td align="left" width="50%">
  <input type="text" name="kaufgebühr" size="10"/> ?
```

Listing 5.15: neu.php (View)

```
</td></tr>
<tr><td align="right" width="50%">
  Verkauf-Gebühren*:
</td><td align="left" width="50%">
  <input type="text" name="verkaufgebühr" size="10"/> ?
</td></tr>
<tr><td align="right" width="50%">
  <input type="submit" name="eingabe" value="      OK      "/>
</td><td align="left" width="50%">
  <input type="submit" name="eingabe" value="Abbrechen"/>
</td></tr>
</table>
</form>
<p>
*: Die Gebühren, die zu erheben wären, wenn man den Aktienbestand jetzt verkaufen
würde.
</p>
<?php require_once("footer.inc.php"); ?>
```

Listing 5.15: neu.php (View) (Forts.)

Wie in jedem anderen Anwendungsfall werden auch hier der Header und der Footer eingebunden. Die Datei selbst besteht aus einer HTML-Tabelle, in der alle Parameter eingegeben werden, die zur Erstellung eines neuen Aktienbestands notwendig sind. Sie könnten zusätzlich dazu clientseitige Gültigkeitsprüfungen – beispielsweise in einem JavaSkript – vornehmen, um den Benutzer frühzeitig auf Fehleingaben hinzuweisen.

Das HTML-Formular der View verweist auf *neu2.php*, die eine Controller-Datei darstellt. Die eingegebenen Daten werden per HTTP-POST-Protokoll übergeben. Abbildung 5.20 zeigt zunächst die Eingabemaske der Datei *neu.php*.

Wenn Sie in Abbildung 5.20 nun auf *OK* oder *Abbrechen* klicken, werden alle eingegebenen Daten per HTTP-POST an die Datei *neu2.php* übertragen. Deren Quellcode ist in Listing 5.16 dargestellt. Dort wird zunächst geprüft, ob Sie die *Abbrechen*-Schaltfläche betätigt hatten. Dies ist noch vor der Einbindung der *header.inc.php* notwendig, da dort bereits die Ausgabe des HTML-Codes zum Client beginnt. In dem Fall, dass Sie den Vorgang abbrechen wollen, werden Sie nämlich über das direkte HTTP-Kommando *Location: start.php* an *start.php* weitergeleitet. Wenn die Ausgabe der HTML-Seite bereits begonnen hat, ist eine solche Weiterleitung nicht mehr möglich.

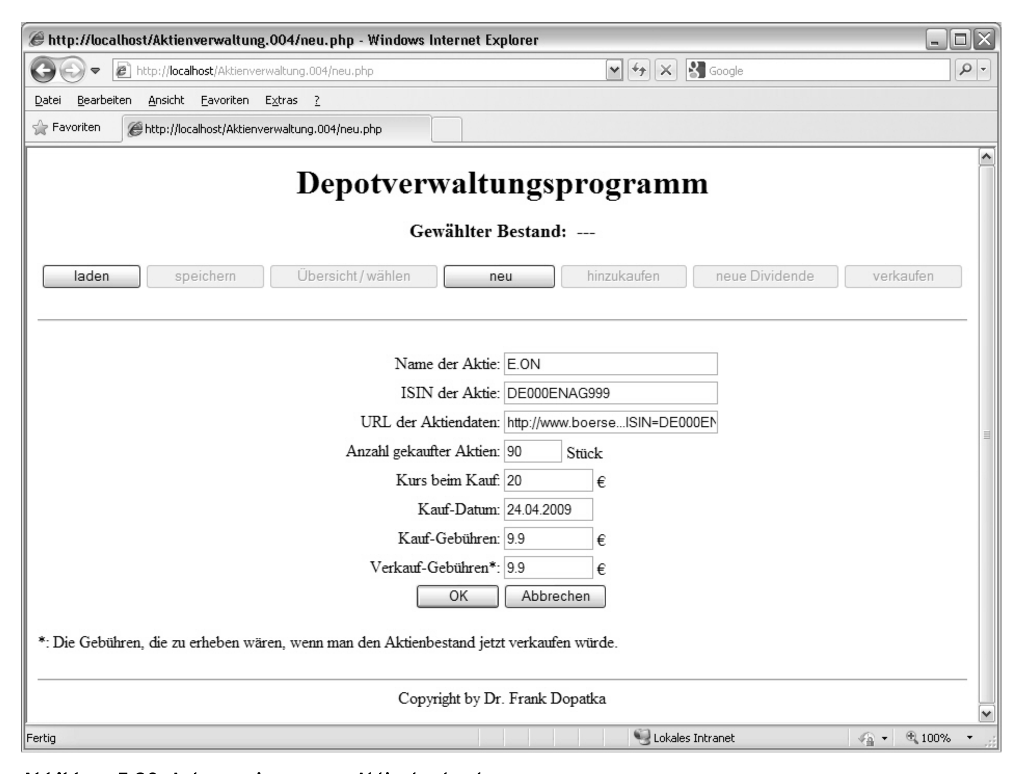


Abbildung 5.20: Anlegen eines neuen Aktienbestands

Im Anschluss an die Einbindung der *header.inc.php* wird aus den eingegebenen Daten des Benutzers, die sich im PHP-Datenfeld *$_POST* befinden, zunächst ein neues Aktienobjekt angelegt (vgl. dazu den ersten Test in Listing 5.1).

> **Hinweis**
>
> Vor dem Anlegen der Aktie müsste eigentlich eine gründliche Prüfung der Inhalte des *$_POST*-Felds vorgenommen werden, selbst wenn dies bereits auf der Seite des Clients geschehen ist. Viele Angreifer versuchen nämlich direkt, manipulierte HTTP-POST-Kommandos auf die Controller-Dateien abzusetzen, um unberechtigten Zugriff auf den Datenbankserver zu erhalten.

Mit der angelegten Aktie und der zuvor in dem GUI eingegebenen Anzahl, dem Datum, Kurs und der Gebühr des Kaufs wird anschließend eine Kauforder erstellt, die wiederum zu einem Aktienbestand führt. Das neu angelegte Bestandsobjekt wird dann in der Session serialisiert und die Anzahl der Bestände wird in der Session erhöht.

Für die Ausgabe der Datei wird abschließend geprüft, ob das Anlegen des Bestands erfolgreich war oder nicht. Dabei wird geprüft, ob der Bestand erfolgreich in der Session gespeichert wurde. Im Erfolgsfall wird der neu angelegte Bestand in der Session ausgewählt und eine Erfolgsmeldung ausgegeben, im Fehlerfall eine Fehlermeldung. In bei-

den Fällen muss die Meldung über die *OK*-Schaltfläche bestätigt werden, die zu der noch zu erstellenden Datei *uebersicht.php* führt.

```
<?php
  if ($_POST[eingabe]=="Abbrechen") header('Location: start.php');
  require_once("header.inc.php");
  $aktie=new Aktie($_POST[name],$_POST[isin],$_POST[url]);
  $kauf=new Kauforder($aktie,$_POST[anz],$_POST[kaufdatum],
                                  $_POST[kaufkurs],$_POST[kaufgebühr]);
  $bestand=new Aktienbestand($aktie,$kauf,$_POST[verkaufgebühr]);
  $_SESSION[Bestand][$_SESSION[BestandAnzahl]]=serialize($bestand);
  $_SESSION[BestandAnzahl]++;
?>
<center>
<h3>
<?php
  if (($bestand!=NULL)&&(unserialize($_SESSION[Bestand][
                              $_SESSION[BestandAnzahl]-1])==$bestand)){
    $_SESSION[BestandGewählt]=$_SESSION[BestandAnzahl]-1;
    echo "Der Bestand wurde dem Depot erfolgreich hinzugefügt!";
  }
  else{
    echo "FEHLER beim Hinzufügen des Bestandes!";
  }
?>
</h3>
<form action="uebersicht.php" method="post"><input type="submit"
                                        value="      OK      "/></form>
</center>
<?php require_once("footer.inc.php"); ?>
```

Listing 5.16: neu2.php (Controller)

Abbildung 5.21 zeigt die Erfolgsmeldung bei einem erfolgreich angelegten Aktienbestand.



Abbildung 5.21: Ausgabe nach dem erfolgreichen Anlegen eines Aktienbestands

Durch das Drücken der *OK*-Schaltfläche wird nun mit dem gerade ausgewählten, neu angelegten Aktienbestand auf *uebersicht.php* zugegriffen. In dieser Datei sollen die Bilanzen aller Bestände angezeigt werden, sodass man einen Überblick über sein Depot erhält. Zusätzlich soll man dort einen Bestand auswählen können, um mit diesem weitere Aktionen durchzuführen, wie Aktien nachzukaufen, eine Dividende hinzuzufügen oder den Bestand zu verkaufen. Ein Großteil dieses Quellcodes ist in Listing 5.17 dargestellt.

```
<?php require_once("header.inc.php"); ?>
<?php
for($i=0;$i<$_SESSION[BestandAnzahl];$i++){
  $bestand=unserialize($_SESSION[Bestand][$i]);
  $bilanz=$bestand->getBilanz();
  echo '<font face="Arial,Helvetica">';
  echo '<b>'.$bilanz[aktienname].'</b>';
  echo ' ('. $bilanz[aktienISIN].')</b>';
  if($bestand->isVerkauft()){
    echo ' &lt;&lt;VERKAUFT&gt;&gt;<br>';
  }
  else{
    echo ' <a href="uebersicht.php?w='.$i.'">wählen...</a><br>';
  }
  echo $bilanz[anzahl].' Stück am '.$bilanz[ekDatum].' gekauft';
  if($bestand->isVerkauft()){
    echo ' und '.number_format($bilanz[tageImBesitz],0).' Tage gehalten';
  }
```

Listing 5.17: uebersicht.php (View)

```
  else{
    echo ' (vor '.number_format($bilanz[tageImBesitz],0).' Tagen)';
    echo '; Stand vom '.$bilanz[aktDatum].', '.$bilanz[aktZeit].'Uhr';
  }
  if ($bilanz[ekSumme]>$bilanz[aktSumme]){
    $farbe="#FF0000";
  }
  else{
    $farbe="#00FF00";
  }
  echo '<table border="0" width="380px">';
  echo '<tr><td>';
  echo '<font face="Arial,Helvetica" size="-1">EK/akt.-Wert gesamt incl.
                                                         Kosten:</font>';
  echo '</td><td align="right">';
  echo '<font face="Arial,Helvetica" size="-1" color="'.$farbe.'">'.
                                  number_format($bilanz[ekSumme],2).'? /
                         '.number_format($bilanz[aktSumme],2).'?</font>';
  echo '</td></tr>';
  echo '<tr><td>';
  echo '<font face="Arial,Helvetica" size="-1">EK/akt.-Kurs
                                                      pro Aktie:</font>';
  echo '</td><td align="right">';
  echo '<font face="Arial,Helvetica" size="-1" color="'.$farbe.'">'.
                            number_format($bilanz[ekEinzelpreis],2).'? /
                  '.number_format($bilanz[aktEinzelpreis],2).'?</font>';
  echo '</td></tr>';
....
  echo '</table>';
  echo '<br><br>';
}
?>
<?php require_once("footer.inc.php"); ?>
```

Listing 5.17: uebersicht.php (View)

Mithilfe der *for*-Schleife werden die Aktienbestände nacheinander aus dem Depot ausgelesen und deserialisiert. Danach wird die Bilanz des jeweiligen Bestands ausgelesen und die Daten der Bilanz mit HTML-Code grafisch aufbereitet.

Falls der Bestand bereits verkauft wurde, kann er nicht mehr für weitere Aktionen ausgewählt werden. In diesem Fall wird der Text *<<VERKAUFT>>* (<<VERKAUFT>>) ausgegeben. Ansonsten wird ein Link dargestellt, der wieder auf *uebersicht.php* verweist. Sie bekommt die ID des Bestands in der Liste der Aktienbestände in

der Variable *w* übergeben. Die Zeile *wählen...* setzt bei einem Klick auf den Link das HTTP-Get-Kommando ab. Je nachdem, ob der Bestand bereits verkauft wurde oder noch nicht, wird die Ausgabe der Bilanz also leicht angepasst.

Wenn der Einkaufspreis zuzüglich aller angefallenen Kosten größer ist als der aktuelle Wert des Bestands bzw. der Verkaufswert, jeweils zuzüglich erhaltener Dividenden und abzüglich der Verkaufsgebühr, haben Sie Verlust erwirtschaftet. In diesem Fall gilt *$bilanz[ekSumme]>$bilanz[aktSumme]* und eine Variable *$farbe* erhält den Hex-Wert *#FF0000* als Zeichenkette. Sie haben also rote Zahlen geschrieben. Im anderen Fall wird die Farbe auf den RGB-Wert (Rot-Grün-Blauanteile) *#00FF00* gesetzt und es ist alles im grünen Bereich.

Jede Bilanz wird dann in einer eigenen HTML-Tabelle ausgegeben, die ausschnittsweise dargestellt wird. Die Gleitkommawerte werden auf zwei Nachkommastellen über den Befehl *number_format($wert,2)* gerundet. Die CSS-Angaben zur Style-Formatierung sind in diesem Beispiel noch fest in die HTML-Tabelle integriert. Gerade bei größeren Projekten sollten die Style-Angaben in externe CSS-Dateien ausgelagert werden. Die HTML-Tabellen verweisen dann lediglich auf einen bestimmten CSS-Stil.

Abbildung 5.22 zeigt die Übersicht über die Aktienbestände. In diesem Fall ist genau ein Bestand vorhanden, dessen Bilanz angezeigt wird. Jeder Bestand kann über den Hyperlink *wählen...* in der PHP-Session ausgewählt werden.

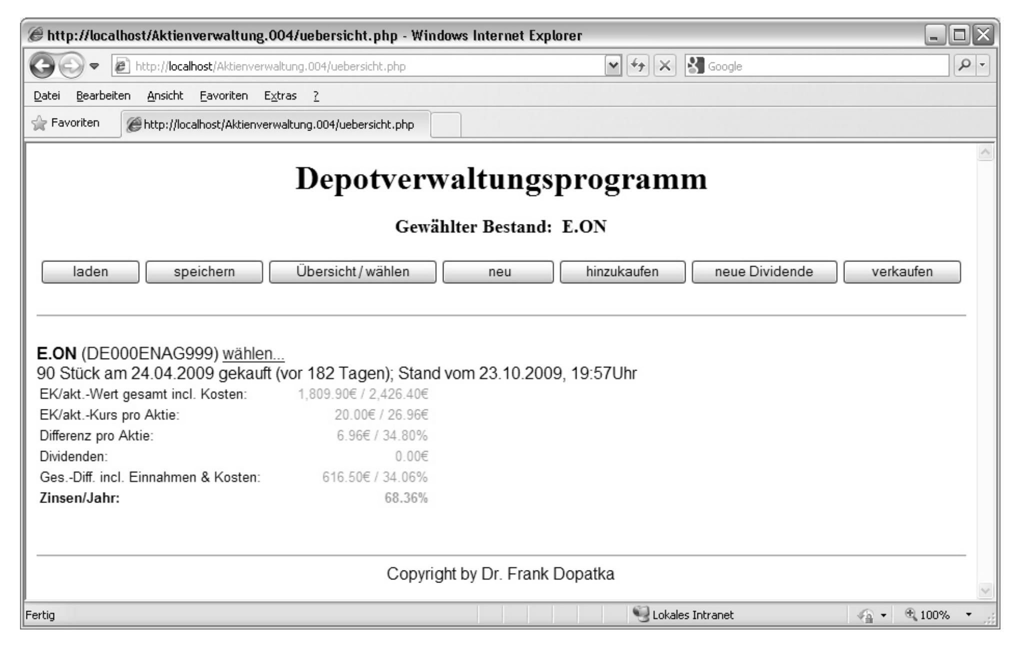


Abbildung 5.22: Ausgabe der Bestandsübersicht

Im nächsten Schritt des Szenarios sollen weitere Aktien zu dem existierenden Bestand hinzugekauft werden, indem der Anwender auf die Schaltfläche *hinzukaufen* klickt.

Aus Sicht der Fachlogik ist ein neuer Kaufvorgang zu erstellen, der in den Bestand integriert wird. Zunächst müssen aber die Daten für die Erstellung des Kaufvorgangsobjekts vorliegen. Diese werden wieder über ein HTML-Formular eingegeben, das in Listing 5.18 dargestellt ist. Das Formular besteht im Wesentlichen aus einer HTML-Tabelle, in der die folgenden Daten eingegeben werden:

- die Anzahl der neu gekauften Aktien (desselben Typs wie in dem Bestand, der gerade ausgewählt wurde)
- der Einzelkurs, zu dem gekauft wurde
- das Datum des Kaufs
- die Gebühren, die bei diesem Kauf entstanden sind

Durch Klick auf die Schaltfläche *Kaufen!* wird abschließend ein HTTP-Post auf die Datei *hinzukaufen2.php* abgesetzt, wobei die eingegebenen Daten übertragen werden.

```
<?php require_once("header.inc.php"); ?>
<form action="hinzukaufen2.php" method="post">
<table border="0" width="100%">

<tr><td align="right" width="50%">
  Anzahl gekaufter Aktien:
</td><td align="left" width="50%">
  <input type="text" name="anz" size="5" value="90"/> Stück
</td></tr>

<tr><td align="right" width="50%">
  Kurs beim Kauf:
</td><td align="left" width="50%">
  <input type="text" name="kaufkurs" size="10" value="23.54"/> ?
</td></tr>

<tr><td align="right" width="50%">
  Kauf-Datum:
</td><td align="left" width="50%">
  <input type="text" name="kaufdatum" size="10" value="24.04.2009"/>
</td></tr>

<tr><td align="right" width="50%">
  Kauf-Gebühren:
</td><td align="left" width="50%">
  <input type="text" name="kaufgebühr" size="10" value="9.9"/> ?
</td></tr>
```

Listing 5.18: hinzukaufen.php (View)

```
<tr><td align="right" width="50%">
  <input type="submit" name="eingabe" value=" Kaufen! "/>
</td><td align="left" width="50%">
  <input type="submit" name="eingabe" value="Abbrechen"/>
</td></tr>
</table>
</form>
<?php require_once("footer.inc.php"); ?>
```

Listing 5.18: hinzukaufen.php (View) (Forts.)

Abbildung 5.23 zeigt die HTML-Datei für die Datenerfassung eines weiteren Aktienkaufs.

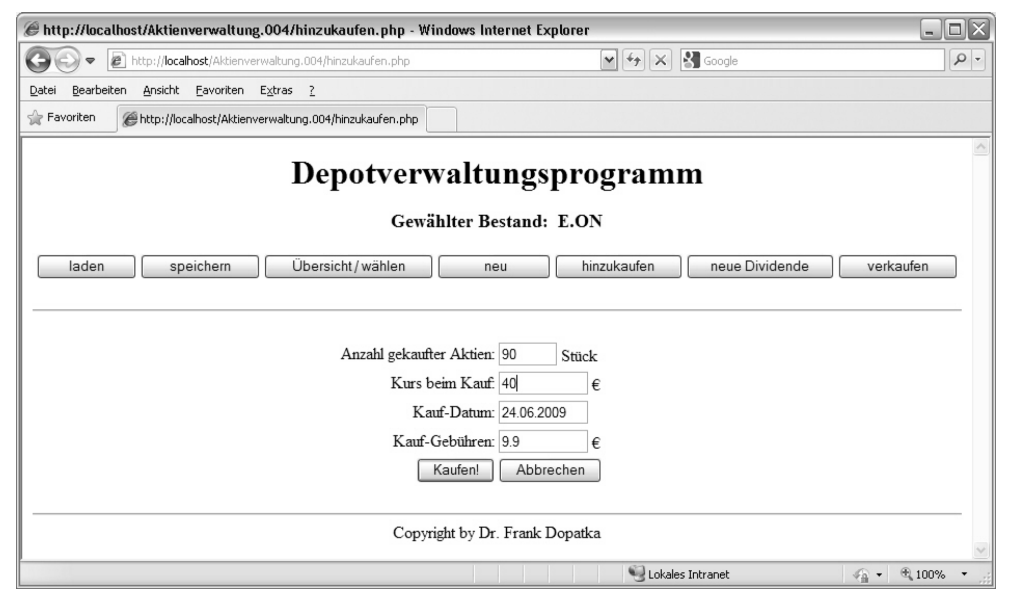


Abbildung 5.23: Eingabemaske für einen weiteren Kauf zu einem Aktienbestand

Die PHP-Datei *hinzukaufen2.php* in Listing 5.19, die den Kaufvorgang in der Fachlogik erfasst, ähnelt in ihrem Aufbau der Datei *neu2.php*, mit der ein neuer Bestand angelegt wurde. Auch hier wurde aus Gründen der Übersicht auf eine Fehlerbehandlung verzichtet.

Zunächst wird wieder ein HTTP-Redirekt durchgeführt, falls der Benutzer sich doch dazu entschieden hat, keinen zusätzlichen Kauf einzugeben. Ansonsten wird wie üblich die
Header-Datei nachgeladen, in der die Session initialisiert wird. Der gewählte Aktienbestand wird dann aus der Session deserialisiert.

Aus der Aktie im gewählten Bestand sowie aus den im HTML-Formular eingegebenen Parametern, die sich nun im PHP-Datenfeld *$_POST* befinden, wird ein neues Kauforderobjekt angelegt und dem Aktienbestand über die Methode *addKauforder* hinzugefügt. Sowohl vor als auch nach dem Hinzufügen wird die Anzahl der Aktien im Bestand abgefragt. Ist die Differenz identisch mit der Anzahl der hinzugekauften Aktien, so kann man davon ausgehen, dass der Kauf erfolgreich registriert wurde. Sowohl im Erfolgsfall als auch im Fehlerfall wird der Benutzer durch die HTML-Ausgabe informiert. Im Erfolgsfall wird zusätzlich das geänderte Aktienbestandsobjekt wieder in der Session serialisiert und damit auf dem Server persistent gehalten.

```
<?php
  if ($_POST[eingabe]=="Abbrechen"){
    header('Location: start.php');
  }
  require_once("header.inc.php");
  $bestand=unserialize($_SESSION[Bestand][$_SESSION[BestandGewählt]]);
  $aktie=$bestand->getAktie();
?>
<center>
<h3>
<?php
  $kauf=new Kauforder($aktie,$_POST[anz],$_POST[kaufdatum],
                                  $_POST[kaufkurs],$_POST[kaufgebühr]);
  $anzalt=$bestand->getAnzahl();
  $bestand->addKauforder($kauf);
  $anzneu=$bestand->getAnzahl();
  if ($anzneu-$anzalt==$_POST[anz]){
    $_SESSION[Bestand][$_SESSION[BestandGewählt]]=serialize($bestand);
    echo "Der Kauf wurde dem Bestand erfolgreich hinzugefügt!";
  }
  else{
    echo "FEHLER beim Hinzufügen der Kaufs zum Bestand!";
  }
?>
</h3>
<form action="uebersicht.php" method="post">
<input type="submit" value="      OK      "/>
</form>
</center>
<?php require_once("footer.inc.php"); ?>
```

Listing 5.19: hinzukaufen2.php (Controller)

Der neue Kauf wurde im Beispiel erfolgreich durchgeführt, sodass es zu der in Abbildung 5.24 dargestellten Meldung für den Benutzer kommt.

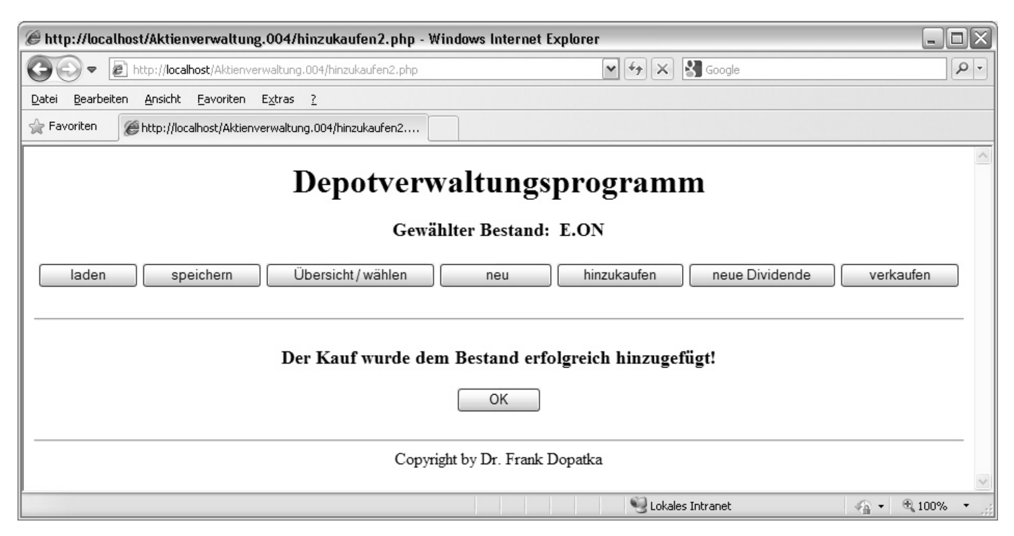


Abbildung 5.24: Bestätigung des erfolgreichen Hinzukaufens zu einem Aktienbestand

Das Klicken auf die *OK*-Schaltfläche führt den Benutzer dann wieder in die Übersicht der Aktienbestände, sodass er ohne weiteres Klicken das Ergebnis seines Zukaufs in der Bilanz des Bestands sehen kann.

Den ersten Kauf konnte der Anwender für günstige 20 €/Aktie tätigen, er hatte damals 90 Stück gekauft. Beim zweiten Kauf hat unser Benutzer ganze 40 €/Aktie bezahlen müssen, wobei er wiederum 90 Stück gekauft hatte. Der durchschnittliche Einkaufspreis liegt also bei 30 €/Stück, wobei zweimalig Kaufgebühren angefallen sind. Die Fachlogik berechnet dabei – wie bereits ohne grafische Benutzerschnittstelle getestet wurde – sowohl den gewichteten Mittelwert des Kurses als auch des Kaufdatums. Dies können Sie in den Abbildungen 5.22, 5.23 und 5.25 nachrechnen.

Bei der Bilanz zieht sich die Fachlogik die aktuellen Kurse von einer Börsenhomepage. Der aktuelle Wert einer Aktie liegt bei 26.96 € und damit unterhalb des mittleren Kaufpreises. Der Aktionär hat also nach dem aktuellen Stand der Dinge Verlust erwirtschaftet, was sich in den roten, negativen Zahlen in Abbildung 5.25 widerspiegelt.

Im nächsten Schritt soll eine Dividende zu dem Bestand hinzugefügt werden, die Aktiengesellschaften typischerweise nach einer Hauptversammlung an die Aktionäre pro Aktie ausschütten. Den Aktionären wird dann der gesamte Dividendenbetrag auf ihr Konto überwiesen, sie sehen also den Gesamtbetrag auf ihrem Kontoauszug. Über unsere Anwendung soll dieser Betrag nun zum Bestand hinzugefügt werden. Die Fachlogik sieht den Betrag als zusätzlichen Gewinn.

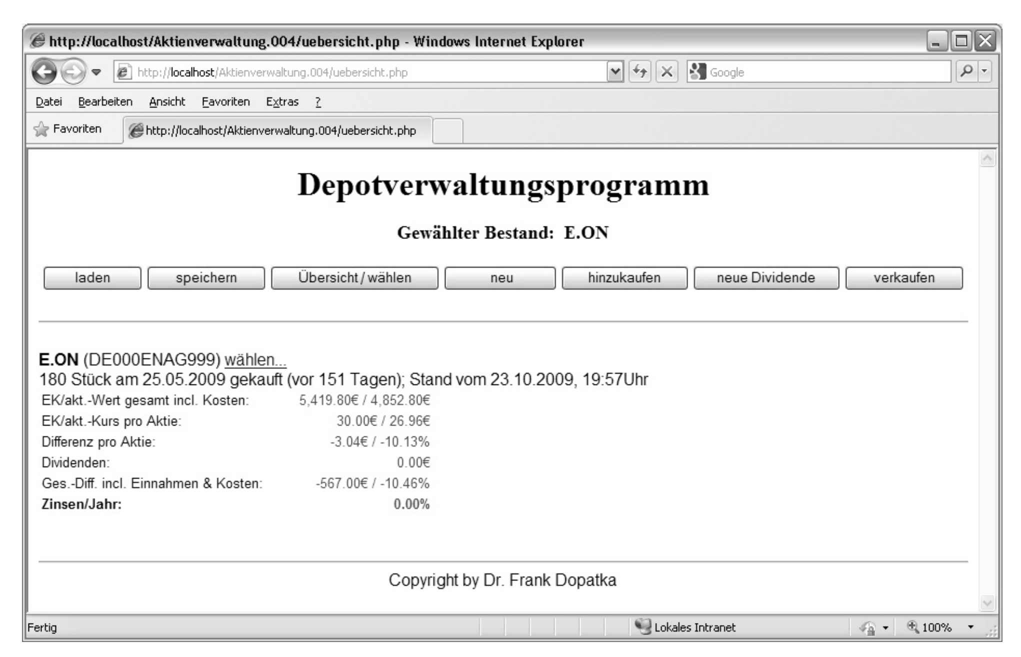


Abbildung 5.25: Neue Bilanz nach dem zweiten Kauf

Listing 5.20 zeigt den HTML-Quellcode für das Formular, um eine neue Dividende einzugeben. Erkennen Sie die Struktur des Quellcodes? Er ist sehr ähnlich dem Quellcode zum Hinzukaufen eines Bestands (Listing 5.18), besteht jedoch vereinfacht nur aus einem Textfeld namens *dividende*, bei dem der Benutzer den Betrag eingeben kann.

```
<?php require_once("header.inc.php"); ?>
<form action="dividende2.php" method="post">
<table border="0" width="100%">
<tr><td align="right" width="50%">
  erhaltene Dividende:
</td><td align="left" width="50%">
  <input type="text" name="dividende" size="10" value=""/> ?
</td></tr>
<tr><td align="right" width="50%">
  <input type="submit" name="eingabe" value="Hinzufügen!"/>
</td><td align="left" width="50%">
  <input type="submit" name="eingabe" value=" Abbrechen "/>
</td></tr>
</table>
</form>
<?php require_once("footer.inc.php"); ?>
```

Listing 5.20: dividende.php (View)

Die entstehende Eingabemaske in Abbildung 5.26 ist dementsprechend kompakt. Durch Klicken der Schaltflächen *Hinzufügen* bzw. *Abbrechen* gelangt der Benutzer per HTTP-Post auf *dividende2.php*.



Abbildung 5.26: HTML-Formular zur Eingabe einer Dividende

Meinung

Versuchen Sie, Ihre Benutzerinteraktionen und HTML-Formulare einfach, verständlich und konsistent über die gesamte Anwendung zu gestalten. Dies sorgt für eine größere Akzeptanz bei den Benutzern und eine angenehme, logische Bedienbarkeit, für die keine zusätzliche Anleitung notwendig ist. Auch komplexe Probleme können stets durch eine strukturierte, teilweise vereinfachte Präsentation für einen Anwender verständlich dargeboten werden.

Der Quellcode des Controllers ist bis zum Auslesen des gewählten Aktienbestands mit dem Quellcode des Hinzufügens eines weiteren Kaufvorgangs aus Listing 5.19 identisch. Im Anschluss daran wird die eingegebene Dividende aus dem *$_POST*-Datenfeld wieder ohne Sicherheitsprüfung entnommen und dem Bestand hinzugefügt. Dabei wird die Summe der erhaltenen Dividenden vor und nach dem Hinzufügen abgefragt, um den Erfolg der Aktion zu überprüfen.

```
<?php
  if (trim($_POST[eingabe])=="Abbrechen"){
    header('Location: start.php');
  }
  require_once("header.inc.php");
  $bestand=unserialize($_SESSION[Bestand][$_SESSION[BestandGewählt]]);
```

Listing 5.21: dividende2.php (Controller)

```
?>
<center>
<h3>
<?php
  $divalt=$bestand->getDividenden();
  $bestand->addDividende($_POST[dividende]);
  $divneu=$bestand->getDividenden();
  if ($divneu-$divalt==$_POST[dividende]){
    $_SESSION[Bestand][$_SESSION[BestandGewählt]]=serialize($bestand);
    echo "Die Dividende wurde dem Bestand erfolgreich hinzugefügt!";
  }
  else{
    echo "FEHLER beim Hinzufügen der Dividende!";
  }
?>
</h3>
<form action="uebersicht.php" method="post">
<input type="submit" value="      OK      "/></form>
</center>
<?php require_once("footer.inc.php"); ?>
```

Listing 5.21: dividende2.php (Controller) (Forts.)

Über den Erfolg wird der Benutzer wie immer benachrichtigt, wie es in Abbildung 5.27 dargestellt ist. Durch das Klicken auf die *OK*-Schaltfläche gelangt er wieder in die Übersicht seiner Bestände, die man als zentrale Ausgabeseite sehen kann.



Abbildung 5.27: Bestätigung des erfolgreichen Hinzukaufens einer Dividende

Die hinzugefügte Dividende erscheint unmittelbar in der Bilanz. Dies zeigt das gute Zusammenspiel der Objekte untereinander sowie die Kooperation der Fachlogik mit den Quellcodes des Benutzerinterfaces der Anwendung.

Außerdem wird durch die zusätzliche Einnahmequelle der Verlust in der Gesamtdifferenz incl. aller Zusatzeinnahmen und Zusatzkosten richtigerweise von -567 € auf -417 € reduziert.

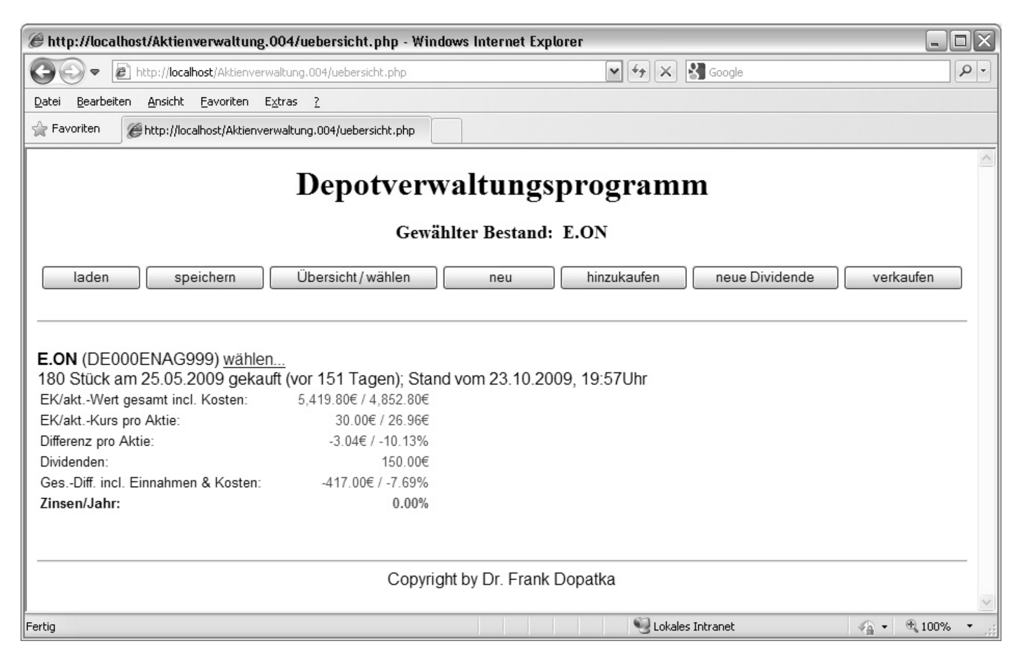


Abbildung 5.28: Reduzierung des Verlusts durch Erhalt der Dividende

Im letzten Schritt des Szenarios soll nun der gesamte Aktienbestand verkauft werden. Dazu muss der Benutzer die Daten des Verkaufs angeben, die in dem HTML-Formular aus Listing 5.22 eingegeben werden. Die Daten werden wie üblich in Textfeldern erfasst, die zusammen mit ihrer Beschreibung in einer HTML-Tabelle angeordnet werden. Dabei handelt es sich um

- den Kurswert der Aktie beim Verkauf
- das Verkaufsdatum
- die angefallenen Gebühren beim Verkauf

```
<?php require_once("header.inc.php"); ?>
<form action="verkaufen2.php" method="post">
<table border="0" width="100%">
<tr><td align="right" width="50%">
  Kurs beim Verkauf:
```

Listing 5.22: verkaufen.php (View)

```
</td><td align="left" width="50%">
  <input type="text" name="verkaufkurs" size="10" value="23.50"/> €
</td></tr>
<tr><td align="right" width="50%">
  Verkauf-Datum:
</td><td align="left" width="50%">
  <input type="text" name="verkaufdatum" size="10" value="28.08.2009"/>
</td></tr>
<tr><td align="right" width="50%">
  Verkauf-Gebühren:
</td><td align="left" width="50%">
  <input type="text" name="verkaufgebühr" size="10" value="9.9"/> €
</td></tr>
<tr><td align="right" width="50%">
  <input type="submit" name="eingabe" value="Verkaufen!"/>
</td><td align="left" width="50%">
  <input type="submit" name="eingabe" value="Abbrechen"/>
</td></tr>
</table>
</form>
<?php require_once("footer.inc.php"); ?>
```

Listing 5.22: verkaufen.php (View)

Abbildung 5.29 zeigt die Eingabemaske für den Verkauf, die sich nahtlos in den Stil der anderen Masken integriert. Sie erkennen bereits, dass zwei typische Arten von HTML-Formularen existieren, nämlich die Eingabemasken sowie die Ausgaben des Controllers. Beide Typen besitzen eine sehr ähnliche HTML-Struktur, die bei Änderungen nur schwer wartbar ist. Daher empfiehlt es sich bei einer größeren Anzahl an Masken, eine eigene View- sowie eine Controller-Klasse zu implementieren, die ein Framework für Eingabemasken bzw. für Auswertungen der Eingabe bilden. Änderungen am Design müssen dann nur in den Frameworkklassen vorgenommen werden und wirken sich auf alle konkreten Masken aus, die aus den Frameworkklassen als GUI-Objekte instanziiert werden.

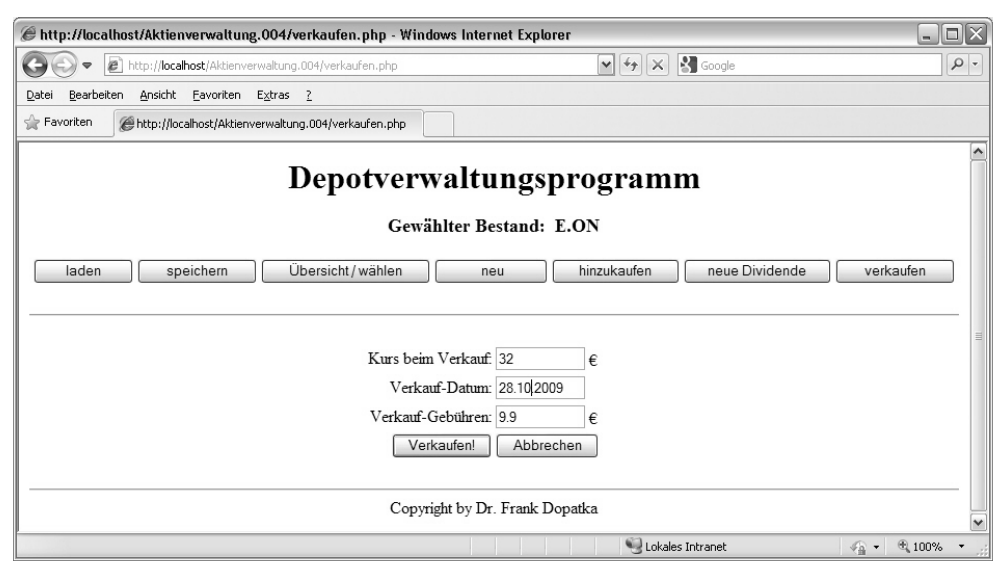


Abbildung 5.29: Eingabemaske zur Erfassung der Verkaufsdaten

Entscheidet sich der Anwender zum Verkauf, wird der gewählte Bestand aus der PHP-Session deserialisiert und aus den eingegebenen Daten eine Verkaufsorder erzeugt, die wiederum dem Bestand hinzugefügt wird. Dadurch wird der Bestand automatisch in der Fachlogik als *verkauft* markiert, was zur Kontrolle abgefragt wird. Dementsprechend wird die Ausgabe für den Benutzer aufbereitet. Im Erfolgsfall wird zusätzlich der verkaufte Bestand wieder in der Session serverseitig serialisiert.

```
<?php
  if ($_POST[eingabe]=="Abbrechen"){
    header('Location: start.php');
  }
  require_once("header.inc.php");

  $bestand=unserialize($_SESSION[Bestand][$_SESSION[BestandGewählt]]);
  $aktie=$bestand->getAktie();
?>
<center>
<h3>
<?php
  $verkauf=new Verkauforder($aktie,$bestand->getAnzahl(),
         $_POST[verkaufdatum],$_POST[verkaufkurs],$_POST[verkaufgebühr]);
  $bestand->addVerkauforder($verkauf);
  if ($bestand->isVerkauft()==TRUE){
    $_SESSION[Bestand][$_SESSION[BestandGewählt]]=serialize($bestand);
```

Listing 5.23: verkaufen2.php (Controller)

```
    echo "Der Bestand wurde erfolgreich verkauft!";
  }
  else{
    echo "FEHLER beim Verkaufen des Bestands!";
  }
?>
</h3>
<form action="uebersicht.php" method="post">
<input type="submit" value="      OK      "/></form>
</center>
<?php require_once("footer.inc.php"); ?>
```

Listing 5.23: verkaufen2.php (Controller) (Forts.)

Die Ausgabe in Abbildung 5.30 dokumentiert den erfolgreichen Verkauf. Die OK-Schaltfläche führt wie immer zur Übersicht.

Da sich der Aktienkurs vor dem Verkauf wieder erholt hat, konnte der Bestand zu 32 €/Aktie wieder verkauft werden. Der durchschnittliche Kaufkurs lag ja bei 30 €/Aktie. Zusätzlich dazu hat der Anleger die Dividende in Höhe von 150 € mitgenommen. Zusammen mit dem Verkauf der 180 Aktien mit einem Gewinn von 2 €/Aktie ergibt sich nach dem Verkauf eine positive Gesamtbilanz.

Der Anleger hat nach Abzug der Kosten in 156 Tagen 480.30 € erwirtschaftet. Dies entspricht einem Gewinn von ca. 9 %. Rechnet man diesen Gewinn linear auf ein volles Jahr um, so ergibt sich ein Jahreszins von über 20 %.



Abbildung 5.30: Erfolgreicher Verkauf eines Aktienbestands

Abschließend ist noch anzumerken, dass der verkaufte Bestand in der Übersicht nicht mehr durch den Link *wählen...* ausgewählt werden kann, da der Benutzer ja keine Aktionen mehr mit einem verkauften Bestand durchführen kann. Die Anzeige in der Übersicht dient also nur noch der statistischen Auswertung. Statt des Links erscheint die Anmerkung <<*VERKAUFT*>> an dem Bestand in Abbildung 5.31.

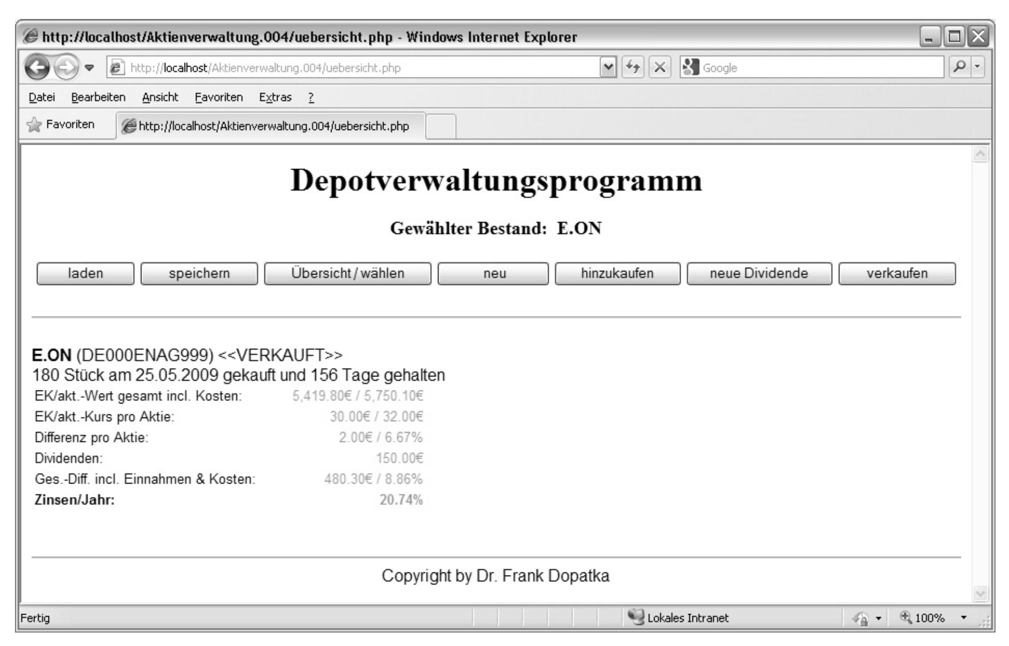


Abbildung 5.31: Die positive Gesamtbilanz der Trading-Aktionen

Der dritte Prototyp: Zugriff auf die Datenbank

Nachdem zunächst die Fachlogik erstellt und getestet wurde, erfolgte im zweiten Schritt das Aufsetzen einer Bedienoberfläche in Form von HTML-Formularen, die serverseitig mit der Fachlogik kommunizieren. Dabei wurden Ansätze des MVC-Paradigmas verwendet. Das im dritten Kapitel vorgestellte Paradigma wurde in den ersten Prototypen noch nicht vollständig und konsequent umgesetzt, da die schnelle Erzeugung eines Prototypen im Vordergrund stand.

Die eingegebenen Daten können bislang in der PHP-Session vorgehalten werden, deren Gültigkeit natürlich zeitlich begrenzt ist. Daher soll im nächsten Schritt eine Datenbank-Anbindung realisiert werden, um die Dreischichtenarchitektur (Kap. 3.1.3) zu vervollständigen.

Ziel ist es also, die aktuellen Zustände der Aktienbestände in der Datenbank festzuhalten und von dort wieder auslesen zu können. Dabei müssen alle Daten berücksichtigt werden, die zur Erstellung der Bilanz notwendig sind. Aus diesen Daten wurde eine MySQL-Datenbank mit Namen *boerse* angelegt, die eine Tabelle *aktien* enthält. Abbildung 5.32 zeigt die Struktur der Tabelle.

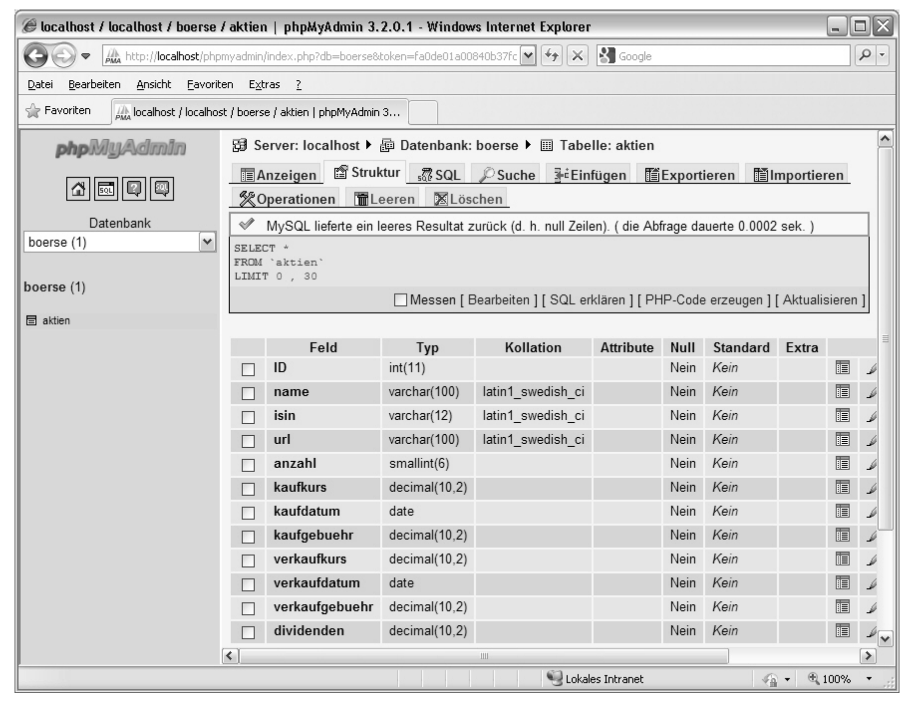


Abbildung 5.32: MySQL-Datenbanktabelle für Aktienbestände

Der Zugriff auf die Datenbank soll wiederum mit den Mitteln der Objektorientierung erfolgen. Dazu wurde bereits in Kapitel 4.2.5 eine Interfacedefinition vorgeschlagen und begründet, mit der ein Zugriff auf eine beliebige Datenbank erfolgen kann. Diese Definition wird in Listing 5.24 nochmals dargestellt.

```
<?php
  interface iDZ{
    public function öffnen($param); public function schliessen();
    public function lesen($param); public function schreiben($param);
  }
?>
```

Listing 5.24: Interface für den Datenbankzugriff

Eine Implementierung des Interfaces wurde für eine MySQL-Datenbank bereits in Listing 4.47 vorgestellt, sodass Sie eine Datenbankverbindung öffnen und schließen sowie lesende und schreibende Zugriffe über SQL-Kommandos durchführen können.

Um die notwendigen Parameter für die Zugriffe zu übergeben, wurde in Listing 4.46 die Hilfsklasse *ParameterListe* vorgestellt und getestet. Diese Klassen wurden zusammen mit dem Interface auf Wiederverwendbarkeit und auf Unabhängigkeit von einer konkreten Problemstellung ausgelegt. Daher können Sie auch für die Persistenzschicht der Depotverwaltung herangezogen werden.

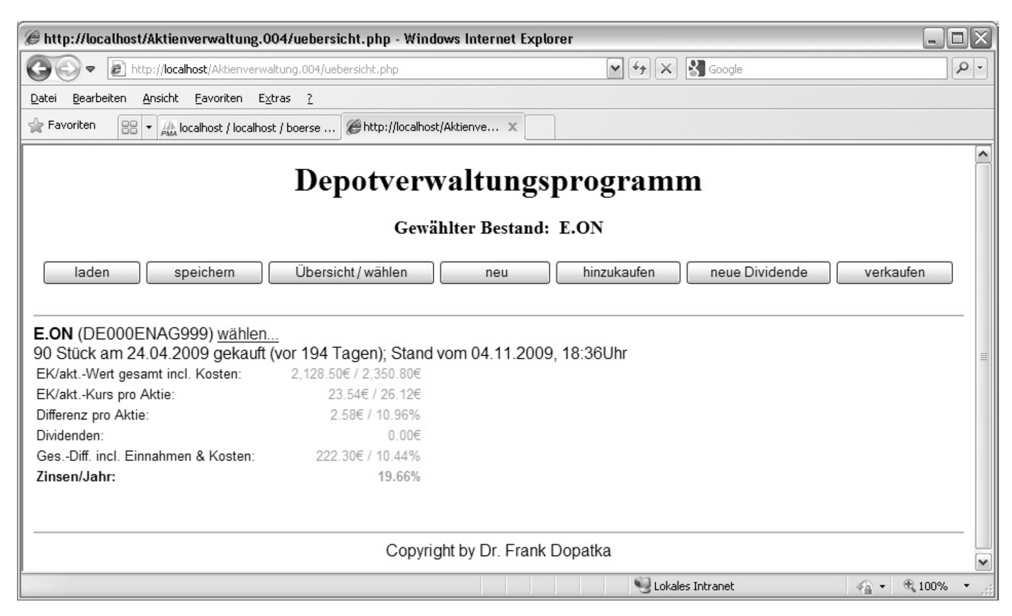


Abbildung 5.33: Ein frisch erstellter Bestand

Ausgangspunkt der Betrachtung ist ein gerade angelegter Datenbestand, der nun persistent (also dauerhaft) in der Datenbank untergebracht werden soll. Dieser Datenbestand ist in Abbildung 5.33 dargestellt. Bei der Erstellung der Menüführung wurde bereits eine Schaltfläche *speichern* vorgesehen, die aus der Anwendungsfallbeschreibung unseres Auftraggebers stammt. Diese Funktion wurde bislang jedoch noch nicht realisiert.

Listing 5.25 beschreibt die Datei *speichern.php*, die bei einem Klick auf die Schaltfläche ausgeführt wird. Dabei wird zuerst geprüft, ob sich überhaupt Aktien im Bestand befinden. Ist dies nicht der Fall, wird eine entsprechende Meldung ausgegeben.

Andernfalls wird ein neues MySQL-Datenbankzugriffsobjekt erzeugt und eine Parameter-Liste gefüllt, die die Zugangsdaten zum Datenbankserver enthält. Im Anschluss daran wird eine Verbindung zum Server aufgebaut.

Ist dies erfolgreich, so wird zunächst ein eventuell bestehender Inhalt in der Tabelle entfernt, um den aktuellen Bestand bzw. die aktuellen Bestände abzulegen. Die Anwendung ist also noch nicht dafür geeignet, Bestände mehrerer Benutzer zu verwalten.

In der anschließenden for-Schleife wird die Liste der Aktienbestände durchlaufen und jeder einzelne Bestand deserialisiert. Dann wird die aktuelle Bilanz des gerade deserialisierten Bestands erstellt, die ja alle notwendigen Informationen enthält. Alle Basisdaten, die zur Erstellung der Bilanz notwendig sind, werden dann in einem SQL-INSERT-Statement verpackt und über ein weiteres Parameterobjekt an die Datenbank gesendet.

War der Schreibvorgang erfolgreich, wird ein Zähler für die erfolgreich archivierten Bestände hochgesetzt, ansonsten ein Zähler für die nicht erfolgreich archivierten Bestände. Abschließend werden beide Zähler im Kontext einer HTML-Ausgabe ausgegeben, sofern sie Werte größer als 0 enthalten.

Die *OK*-Schaltfläche führt dann wieder auf die Übersicht der Bestände, die in der *uebersicht.php* aufbereitet werden.

```
<?php require_once("header.inc.php"); ?>
<center>
<h3>
<?php
  if ($_SESSION[BestandAnzahl]<1){
    echo 'Es sind KEINE Daten zum Speichern vorhanden!';
  }
  else{
    $db=new mysqlDZ();
    $p_öffnen=new ParameterListe();
    $p_öffnen->add('host','localhost'); $p_öffnen->add('user','root');
    $p_öffnen->add('pass',''); $p_öffnen->add('db','boerse');
    if ($db->öffnen($p_öffnen)==FALSE){
      echo 'FEHLER beim Öffnen der Datenbank!';
    }
    else{
      $p_schreiben=new ParameterListe();
      $p_schreiben->add('sql','TRUNCATE aktien');
      $db->schreiben($p_schreiben);
      $erfolg=0;
      $fehler=0;
      for ($i=0;$i<$_SESSION[BestandAnzahl];$i++){
        $bestand=unserialize($_SESSION[Bestand][$i]);
        $data=$bestand->getBilanz();
        $p_schreiben=new ParameterListe();
        $sql="INSERT INTO aktien VALUES (";
        $sql.=($i+1).",'".$data[Aktienname]."','".$data[AktienISIN].
                                            "','".$data[AktienURL]."'";
        $sql.=",".$data[anzahl].",".$data[ekEinzelpreis].
                    ",'".date_format(new DateTime($data[ekDatum]),"Y-m-d").
                                                   "',".$data[ekGebühr];
        if ($bestand->isVerkauft()){
          $sql.=",".$data[aktEinzelpreis].",'".date_format(
            new DateTime($data[aktDatum]),"Y-m-d")."',".$data[aktGebühr];
        }
        else{
          $sql.=",0,'0000-00-00',0";
        }
        $sql.=",".$data[dividenden].")";
        $p_schreiben->add('sql',$sql);
        if ($db->schreiben($p_schreiben)==FALSE)
          $fehler++;
        else
```

Listing 5.25: Speichern der Aktienbestände der PHP-Session

```
          $erfolg++;
        }
        $db->schliessen();
      }
    }
    if ($erfolg>0){
      echo 'Es wurden '.$erfolg.' Datensätze erfolgreich geschrieben.<br>';
    }
    if ($fehler>0){
      echo 'Das Schreiben von '.$fehler.' Datensätzen ist
                                                  FEHLGESCHLAGEN.<br>';
    }
?>
</h3>
<form action="uebersicht.php" method="post">
<input type="submit" value="      OK      "/></form>
</center>
<?php require_once("footer.inc.php"); ?>
```

Listing 5.25: Speichern der Aktienbestände der PHP-Session (Forts.)

In unserem Beispiel erfolgt die Meldung, dass ein Datensatz erfolgreich archiviert wurde.



Abbildung 5.34: Meldung nach erfolgreicher Speicherung

Hinweis

Testen Sie einen Fehlerfall, indem Sie den MySQL-Server im XAMPP Control Panel deaktivieren, bevor Sie auf die Schaltfläche *speichern* klicken.

Den Erfolg können Sie prüfen, indem Sie über phpMyAdmin nochmals die Verwaltung des Datenbankservers aufrufen und sich dort den Inhalt der Aktientabelle anzeigen lassen.

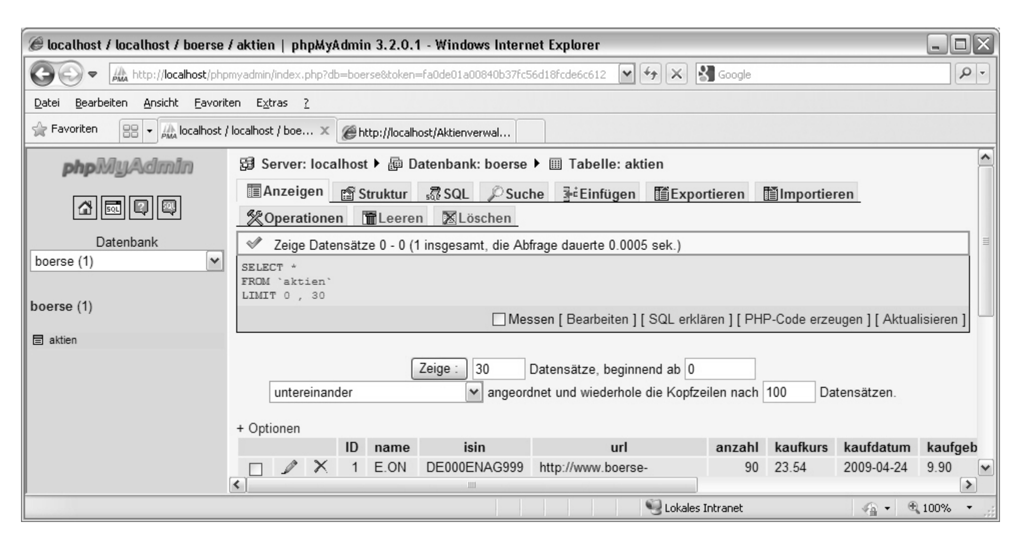


Abbildung 5.35: Inhalt der Datenbanktabelle nach erfolgreicher Speicherung

Im zweiten Teil soll jetzt der bestehende Datenbestand wieder ausgelesen werden. Dazu werden alle Fenster des Internetbrowsers beim Client geschlossen, um die PHP-Session zu beenden. Nach einem erneuten Öffnen erscheint dann wieder der in Abbildung 5.36 dargestellte Willkommensbildschirm.



Abbildung 5.36: Willkommensbildschirm nach neuer Session

Es wurde bereits programmiert, dass die Schaltfläche *speichern* inaktiv ist, wenn keine Bestände vorliegen. Statt neue Bestände manuell anzulegen, sollen Sie über die Schaltfläche *laden* auch existierende Bestände aus der Datenbank einlesen können.

Die Funktion der *laden.php*, die durch das Klicken auf die gleichnamige Schaltfläche ausgeführt wird, ist in Listing 5.26 beschrieben. Wie schon beim Speichern, wird ein neues Datenbankzugriffsobjekt erstellt und die Verbindung zur Datenbank über die Angabe der Parameter *host*, *user*, *passt* und *db* hergestellt.

Über den SQL-Befehl *SELECT * FROM aktien ORDER BY ID* werden dann alle Bestände aus der Datenbanktabelle ausgelesen. Ist dies erfolgreich, so ergibt sich eine mit den Beständen gefüllte Ergebnismenge *$ausgabe*. Diese Menge wird in der foreach-Schleife Datensatz für Datensatz und damit Bestand für Bestand durchgegangen.

Für jeden Bestand wird dann ein Kauforderobjekt erstellt, mit dessen Hilfe ein neues Bestandsobjekt angelegt wird. Dann werden eventuell gezahlte Dividenden zum Bestand hinzugefügt. Wurde der Bestand bereits verkauft, so wird anschließend ein Verkaufsorderobjekt erstellt und dem Bestand hinzugefügt. Dies sind dieselben Schritte, die auch ein Anwender in der Benutzeroberfläche tätigen kann. Es werden dabei dieselben Methoden der Fachlogik aufgerufen. Die Daten stammen diesmal lediglich aus der Datenbank und nicht aus gefüllten HTML-Formularen.

Der Bestand wird abschließend in die PHP-Session serialisiert und der Zähler der Objekte in der Session inkrementiert. Der Benutzer enthält dann als HTML-Antwort die Anzahl der erfolgreich ausgelesenen Aktienbestände.

```
<?php require_once("header.inc.php"); ?>
<center>
<h3>
<?php
  $db=new mysqlDZ();
  $p_öffnen=new ParameterListe();
  $p_öffnen->add('host','localhost'); $p_öffnen->add('user','root');
  $p_öffnen->add('pass',''); $p_öffnen->add('db','boerse');
  if ($db->öffnen($p_öffnen)==FALSE){
    echo 'FEHLER beim Öffnen der Datenbank!';
  }
  else{
    $p_lesen=new ParameterListe();
    $p_lesen->add('sql','SELECT * FROM aktien ORDER BY ID');
    $ausgabe=$db->lesen($p_lesen);
    if ($ausgabe==FALSE){
      $db->schliessen();
      echo 'FEHLER beim Zugriff auf die Datenbank!';
    }
```

Listing 5.26: Laden der Aktienbestände in die PHP-Session

```
    else{
      $i=0;
      foreach($ausgabe as $index => $data){
        $aktie=new Aktie($data[name],$data[isin],$data[url]);
        $kauf=new Kauforder($aktie,$data[anzahl],date_format(
                 new DateTime($data[kaufdatum]),"d.m.Y"),$data[kaufkurs],
                                                    $data[kaufgebuehr]);
        $bestand=new Aktienbestand($aktie,$kauf,$data[verkaufgebuehr]);
        $bestand->addDividende($data[dividenden]);
        if ($data[verkaufkurs]>0){
          // bereits verkauft
          $verkauf=new Verkauforder($aktie,$data[anzahl],date_format(
           new DateTime($data[verkaufdatum]),"d.m.Y"),$data[verkaufkurs],
                                                 $data[verkaufgebuehr]);
          $bestand->addVerkauforder($verkauf);
        }
        $_SESSION[Bestand][$i]=serialize($bestand);
        $i++;
      }
      // Anzahl der Datensätze
      $_SESSION[BestandAnzahl]=count($ausgabe);
      if (count($ausgabe)>0){
        $_SESSION[BestandGewählt]=0;
        echo 'Es wurden '.count($ausgabe).' Aktienbestände geladen.';
      }
      else{
        $_SESSION[BestandGewählt]=-1;
        echo 'ACHTUNG: Es wurde KEIN Bestand geladen.';
      }
      $db->schliessen();
    }
  }
?>
</h3>
<form action="uebersicht.php" method="post">
<input type="submit" value="      OK      "/></form>
</center>
<?php require_once("footer.inc.php"); ?>
```

Listing 5.26: Laden der Aktienbestände in die PHP-Session (Forts.)

Abbildung 5.37 zeigt, dass in diesem Beispiel genau ein Bestand ausgelesen wurde, der auch direkt gewählt wird.



Abbildung 5.37: Erfolgsmeldung nach erfolgreichem Laden

Der Klick auf die *OK*-Schaltfläche führt wie immer in die Übersicht, sodass der Benutzer die Bilanz des aus der Datenbank geladenen Aktienbestands betrachten kann.

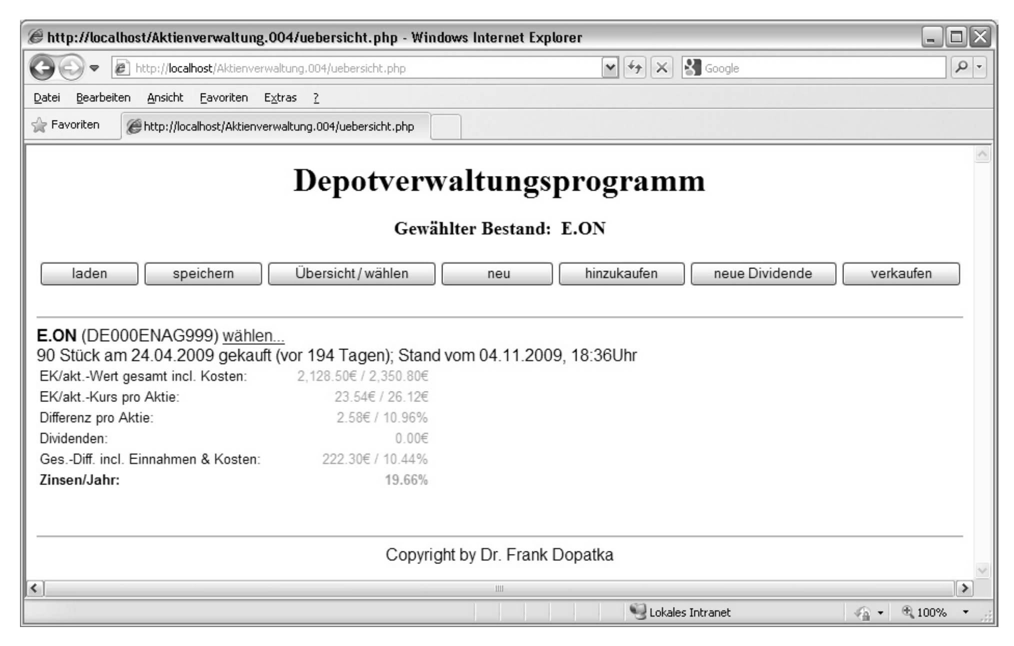


Abbildung 5.38: Der geladene Aktienbestand

5.1.5 Die nächsten Schritte

In diesem Kapitel haben Sie bisher erfahren, wie Sie auf der Grundlage einer kurzen fachlichen, objektorientierten Analyse mit anschließendem technischen Design einen

PHP-Prototypen erstellen, der über die MySQL-Datenzugriffsschicht und die objektorientierte Fachlogik bis hin zu einem Bedieninterface auf Grundlage von HTML-Formularen reicht. Dies entspricht einem Zyklus in der iterativ-inkrementellen Softwareentwicklung.

Nachdem Sie das Vorgehen von der Idee unseres Auftraggebers bis hin zu der Fertigstellung dieses Prototyps gelesen und verstanden haben, können Sie Ihre ursprüngliche Einschätzung für den Aufwand dieser Entwicklung mit der tatsächlichen Dauer bzw. mit den tatsächlichen Kosten vergleichen. Lagen Sie in Ihrer Einschätzung richtig? Hatten Sie mehr oder weniger Zeit bzw. Kosten vorgesehen? In der Regel kalkuliert man als Anfänger weniger Zeit bzw. Kosten als tatsächlich benötigt werden.

Wie könnten nun die nächsten Schritte in dem Szenario dieses Entwicklungsprozesses aussehen? Zunächst einmal ist spätestens nun ein Review durch den Auftraggeber anzusetzen mit den folgenden Fragestellungen:

- Wurden die realisierten Funktionen fachlich korrekt realisiert?
- Wie sind Sie mit der Bedienung zufrieden?
- Was wünschen Sie sich als Nächstes für Funktionen?

Auf Basis der Antworten kann eine Kostenabschätzung und eine Priorisierung für die nächste Iteration vorgenommen werden. Diese bislang realisierten Funktionen können dem Kunden bereits in Rechnung gestellt werden. Für den Fall, dass Sie den Aufwand falsch eingeschätzt haben, ist Ihr Risiko bei dieser Vorgehensweise geringer als bei einer Gesamtabschätzung.

Außerdem könnte ein Refactoring für die bisherige Anwendung erfolgen. Dabei sind drei Punkte zu diskutieren:

1. Die Klassen *Bestand* und *Bilanz* sind sehr eng miteinander verbunden. Es könnte sinnvoller sein, diese beiden Klassen zu vereinigen.
2. Wenn der Kunde davon ausgeht, dass diese Anwendung noch wesentlich größer wird, sollte über die Erstellung von Template-Klassen für die Darstellung (View) und für den Controller nachgedacht werden. Dies führt zu einer verbesserten Wartbarkeit der Gesamtanwendung an der Benutzerschnittstelle.
3. An vielen Stellen werden Benutzereingaben noch nicht auf Gültigkeit geprüft. Ebenso ist die Datenbankanbindung idealisiert und kann eine Vielzahl von Fehlern verursachen. An dieser Stelle kann durch das Erstellen eigener Exception-Klassen ein Fehlermanagement eingeführt werden, das in die Bedienung integriert wird. Zusätzlich dazu könnten Benutzereingaben clientseitig über JavaScript-Prüfroutinen verifiziert werden, noch bevor das HTML-Formular zum Server zurückgesendet wird.

Zusätzlich dazu sind bei der Entwicklung des ersten Prototyps einige Erweiterungsmöglichkeiten aufgefallen, die in den nächsten Iterationen realisiert werden könnten. Dies sind:

- Die Erstellung eines Logins mit Benutzernamen und Kennwort sowie eine benutzerabhängige Speicherung der Aktienbestände. So hat jeder Aktionär Zugriff auf seine persönlichen Bestände. Diese Idee hat bereits unser Auftraggeber bei der Analyse geäußert (Abb. 5.1). Er hat die Umsetzung jedoch damals zurückgestellt.

- Die Anbindung eines realen Depots im Onlinebanking. Die PHP-Anwendung würde dann für den Depotbesitzer einen Mehrwert durch die Bilanzierung darstellen.
- Die Integration mehrerer Depots für einen Benutzer. Somit könnten Aktienbestände gruppiert werden.
- Die Funktion, auch Teile eines Bestands zu verkaufen. Die dabei erzielten Gewinne bzw. Verluste würden dann separat in der Bilanz verwaltet.
- Eine grafische Statistik, die eine Historie des Depotwerts anzeigt über einen Tag, eine Woche, einen Monat und ein Jahr.
- Die Verwaltung laufender Depotkosten. Manche Depotanbieter verlangen zusätzlich zu den Transaktionsgebühren jährliche Gebühren für die Verwaltung eines Depots. Diese Gebühren reduzieren natürlich den Gewinn und sollten in die Bilanz mit eingepflegt werden können.
- Die Möglichkeit, einzelne Bestände in die Datenbank zu laden und zu speichern. Bislang ist nur ein Laden und Speichern der gesamten Bestände möglich.

Hinweis

Als Übung können Sie den bislang vorgestellten Prototyp realisieren und sich im Anschluss daran eine oder mehrere der oben skizzierten Funktionen vornehmen. Schätzen Sie den Aufwand für die Realisierung der Funktionen und binden Sie die Funktionalität in die Anwendung mit ein. Wie präzise können Sie die Aufwandschätzungen vornehmen? Wie nahe liegen Sie an den benötigten Ressourcen der realen Umsetzung?

5.2 Erstellung von gutem Quellcode

Abschließend werden in diesem Buch Regeln und Werkzeuge vorgestellt, die bei der Entwicklung einer PHP-Anwendung behilflich sind und zu einer Verbesserung der Qualität der erstellten Skripte beitragen sollen.

5.2.1 Ein Styleguide für guten Quellcode

Im ersten Schritt werden dabei Regeln für die Erstellung von PHP-Quellcode aufgestellt. Diese Regeln müssen nicht zwingend eingehalten werden (Sie können auch ohne diese Regeln lauffähige PHP-Skripte schreiben), jedoch hat die Praxis gezeigt, dass gerade bei einer größeren Anzahl an Skripten die Wartung und Fehlerkorrektur wesentlich schwieriger wird. Dies gilt auch für den Fall, dass sich andere Entwickler in Ihren Quellcode einarbeiten müssen. Im Folgenden werden 16 „goldene Regeln“ für guten PHP-Quellcode vorgestellt:

1. Schreiben Sie für jede PHP-Klasse eine Datei, die genauso heißt wie die Klasse selbst. Dies erleichtert die Übersicht im Dateisystem und ermöglicht die Programmierung eines automatischen Klassenladers. Dies ist auch bei anderen Sprachen wie Java ver-

breitet. Da Klassen immer von anderen Skripten eingebunden werden, sollte die Dateiendung einer Klasse stets *.inc.php* lauten.

2. Schreiben sie den ersten Buchstaben des Klassennamens stets groß, sowohl im Dateinamen als auch in der Klassendefinition. Erstellen Sie also eine *Aktie.inc.php* mit der Deklaration *class Aktie{...}*. Verwenden Sie stets die Singularform bei der Namensgebung.
3. Schreiben Sie den ersten Buchstaben von Eigenschaften und Methoden stets klein. Wenn es sich um ein zusammengesetztes Wort handelt, schreiben Sie zur besseren Lesbarkeit die ersten Buchstaben der Teilwörter groß, beispielsweise *$farbe* oder *$aktienListe* für die Bezeichnung der Eigenschaften und *berechneGewinn()* für eine Methode.
4. Deklarieren Sie alle Eigenschaften als *private* zur Datenkapselung und alle Methoden, die Dienste eines Objekts dieser Klasse darstellen, als *public*.
5. Schreiben Sie zu jeder Eigenschaft eine entsprechende Get- und Set-Methode. So gehören in einer Stifteklasse beispielsweise zu der Eigenschaft *$farbe* die Methoden *getFarbe()* und *setFarbe($f)*. Prüfen Sie innerhalb der Set-Methode, ob der übergebene Wert *$f* gültig ist und im aktuellen Zustand des Objekts gesetzt werden darf. In der Get-Methode können Sie eine Formatierung für die Ausgabe vorsehen.
6. Sind Eigenschaften schreibgeschützt, so setzen Sie die Set-Methode *private* und greifen nur innerhalb der Klasse zu. Dürfen Eigenschaftswerte (z. B. aus Datenschutzgründen) nicht ausgelesen werden, so setzen Sie die Get-Methode *private*.
7. Fragen Sie einen Wahrheitswert ab, so schreiben Sie nicht die Methode *getVerkauft()*, sondern *isVerkauft()*. Dies macht dem Benutzer dieser Klasse, der ja auch ein Programmierer sein kann, die Handhabung leichter.
8. Wenn Sie einen Wert zu einer kumulierenden Eigenschaft hinzufügen wollen, schreiben Sie nicht *setDividende($d)*, sondern *addDividende($d)*. Dies verdeutlicht, dass etwas aufaddiert wird und gilt auch für Objekte. Meldet sich ein Student zu einem Praktikum an, deklarieren Sie im Praktikum die Methode *addStudent($s)*.
9. Schreiben Sie Konstanten immer komplett groß. Trennen Sie zusammengesetzte Wörter in Konstanten stets mit einem Unterstrich. Beispiele sind *PI* oder *MAX_STUDENTEN*.
10. Eine Klasse hat prinzipiell den folgenden Aufbau:
    - (statische) Konstanten
    - Klasseneigenschaften
    - Eigenschaften jedes Objekts
    - Konstruktoren
    - alle Get- und Set-Methoden der Eigenschaften, jeweils *Get* und *Set* zu jeder Eigenschaft abwechselnd
    - alle öffentlichen Dienste bzw. Methoden dieser Klasse
    - alle privaten Hilfsmethoden
11. Rücken Sie jeden Funktionsblock, z. B. innerhalb einer Klasse, einer Methode, einer if-Verzweigung oder einer Schleife einheitlich mit zwei Leerzeichen ein.

12. Klammern Sie Funktionsblöcke stets einheitlich. Dabei hat sich folgendes System bewährt:
```
class Test{
 $eigenschaft;
 methode(){
  anweisung;
  if ($var==true){
   anweisung;
   anweisung;
  }
 }
}
```
13. Deklarieren Sie PHP-Zeichenketten stets mit einfachen Hochkommata, also *$var='Hallo';*. Dies beschleunigt die Verarbeitung und ist besonders dann hilfreich, wenn Sie den Inhalt der Variablen für eine HTML-Ausgabe verwenden wollen. Dort können Sie dann nämlich doppelte Hochkommata verwenden, ohne mit der PHP-Deklaration zu kollidieren. Ein Beispiel dafür ist *$var='<table border="1">';*.
14. Es gibt prinzipiell zwei Möglichkeiten, HTML-Code in PHP einzubinden. Entweder beenden Sie den PHP-Block und geben HTML-Code aus, oder Sie geben den HTML-Code unter Verwendung des PHP-Befehls *echo* zurück. Die erste Methode hat den Vorteil, dass Sie HTML-Code direkt aus einem HTML-Bearbeitungsprogramm wie Microsoft Frontpage oder Macromedia Dreamweaver einbinden können und der HTML-Code nicht so stark von PHP-Anweisungen durchsetzt wird. Hier sehen Sie ein Beispiel dazu:
```
<? php
 if ($var==true){
  ?>
   <h1>Hallo</h1>
   <p> Herzlich Willkommen auf meiner Homepage</p>
  <?php
 }
?>
```
15. Schreiben Sie vor jeder Klasse einen Kommentar, der Informationen zu der Klasse, ihrem Autor und ggf. Copyrights beinhaltet. Kommentieren Sie vor jeder Methode, was diese Methode dem Benutzer an Funktionalität anbietet.
16. Trennen Sie Skripte mit Fachlogik von Skripten mit Zugriff auf die Datenbank und Skripten für die Benutzerinteraktion. Arbeiten Sie strikt nach dem Model-View-Controller-Prinzip (MVC). Auch wenn dies anfangs unhandlicher erscheint, ergibt diese Vorgehensweise eine bessere Struktur Ihrer Anwendung. Sie können zur Unterscheidung die Namen der Skripte auch entsprechend benennen, z. B. *m_xxxx.php* für Zugriffe auf die Datenbank, *v_xxxx.php* für HTML-Formulare bzw. HTML-Ausgaben sowie *c_xxxx.php* für Skripte, die Fachlogikklassen verwenden.

5.2.2 Erfolgreiche Codestrukturen – Design Patterns

Neben den Regeln für eine bessere Lesbarkeit des Quellcodes haben sich auch Regeln gefunden, wie man bestimmte Probleme objektorientiert lösen kann. Ein Design Pattern (Entwurfsmuster) beschreibt eine bewährte Schablone für ein Entwurfsproblem und damit eine wieder-verwendbare Vorlage zur Problemlösung.

Der Nutzen eines Design Patterns liegt in der Beschreibung einer Lösung für eine bestimmte Klasse von Entwurfsproblemen, die einem Entwickler immer wieder über den Weg laufen. Die Design Patterns wurden erstmals von Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides in ihrem Buch „Design Patterns – Elements of Reusable Object-Oriented Software" erwähnt. Dieses Buch hat einen Standard in der Softwaretechnik geschaffen; seine Autoren sind seitdem als „Viererbande" bzw. „Gang-of-Four" bekannt.

Design Patterns werden heutzutage an jeder deutschen Hochschule im Rahmen der Informatikausbildung gelehrt, damit die Absolventen diese Muster als Anleitung zur objektorientierten Lösung von typischen Problemstellungen verwenden können.

Die Autoren des Buches klassifizieren ihre Design Patterns nach den beiden Kriterien des Zwecks und des Anwendungsbereichs, auf den sie wirken.

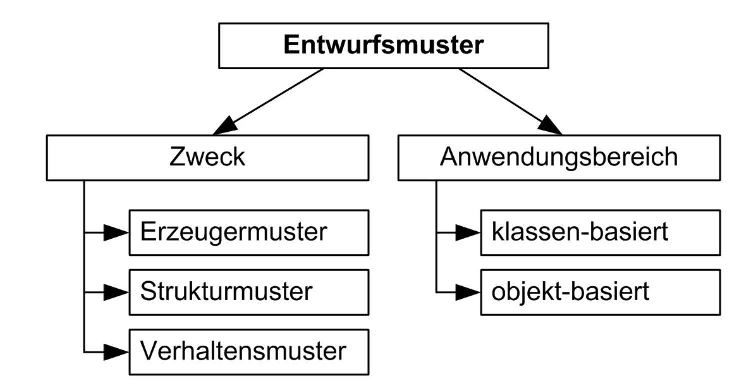
Nach dem Zweck des jeweiligen Musters unterscheiden sie drei Gruppen. Die erste Gruppe der Erzeugungsmuster bezieht sich auf die Erzeugung von Objekten. Ein Anwendungsfall besteht darin, die Anzahl von erzeugten Objekten einer Klasse zu kontrollieren oder den konkreten Typ der erzeugten Objekte anzupassen. Die zweite Gruppe umfasst Strukturmuster, die eine Vereinfachung der Struktur zwischen Klassen ermöglichen sollen. Komplexe Beziehungen können unter anderem über vermittelnde Klassen oder Schnittstellen vereinfacht werden. Die dritte Gruppe der Verhaltensmuster betrifft das Verhalten der Klassen, die sich auf die Zusammenarbeit und den Nachrichtenaustausch von Klassen beziehen.

Nach ihrem Anwendungsbereich lassen sich Muster in zwei Gruppen einteilen. So beschreiben klassenbasierte Muster Beziehungen zwischen Klassen und bauen beispielsweise Vererbungsstrukturen auf. Im Gegensatz dazu nutzen objektbasierte Muster zumeist Assoziationen und Aggregationen zur Beschreibung von Beziehungen zwischen Objekten.

Abbildung 5.39 verdeutlicht die Unterteilung der Design Patterns. Seit ihrer Entdeckung wurden unzählige Design Patterns gemäß dieser Unterteilung von Softwaretechnikern erfunden und veröffentlicht und es wurden einige Bücher zu dieser Thematik geschrieben. Die ursprünglichen Muster der Viererbande haben jedoch als Einzige weltweit Verbreitung und Anerkennung gefunden.

Da es sich bei diesem Buch nicht um ein reines Buch zu Design Patterns handelt, die Design Patterns jedoch in der objektorientierten Entwicklung eine große Bedeutung haben, werden die bekanntesten Vertreter jeder Kategorie im Folgenden vorgestellt. Dabei wird die Problemstellung und der Lösungsansatz jeweils kurz beschrieben, den das entsprechende Design Pattern bietet. Zusätzlich werden UML-Diagramme gezeigt, die den Lösungsansatz genauer beschreiben. Da Sie im dritten Kapitel die UML-Syntax kennengelernt haben und im vierten Kapitel dieses Buches erfahren haben, wie man die

UML-Syntax mit PHP 5 umsetzt, sollten Sie in der Lage sein, diese Muster mit PHP zu realisieren.



	Erzeugermuster	Strukturmuster	Verhaltensmuster
klassen-basiert	Fabrik	Adapter	Schablone
objekt-basiert	Singleton	Kompositum	Beobachter

Abbildung 5.39: Gliederung der Design Patterns und exemplarische Vorstellung

Eine Fabrik zur Produktion von Objekten

Häufig haben Sie die Aufgabenstellung, ein Framework für Anwendungen zu programmieren, die mehrere Dokumente gleichzeitig anzeigen bzw. verwalten können. So können Sie beispielsweise in Microsoft Word neben *.doc*-Dateien auch *.html*- oder *.txt*-Dateien laden, die völlig anders aufgebaut sind. In der Zukunft können weitere, heute noch unbekannte Protokolle hinzukommen.

Das Framework verwendet dabei eine abstrakte Klasse für die Dokumente. Die Erzeugung der Objekte erfolgt durch eine abstrakte Fabrikmethode, die von einer konkreten Unterklasse überschrieben wird. Die Unterklasse (hier: *MeinDokument*) kennt ja ihren Aufbau. Abbildung 5.40 skizziert die notwendige Klassenstruktur zur Realisierung des Fabrikmusters.

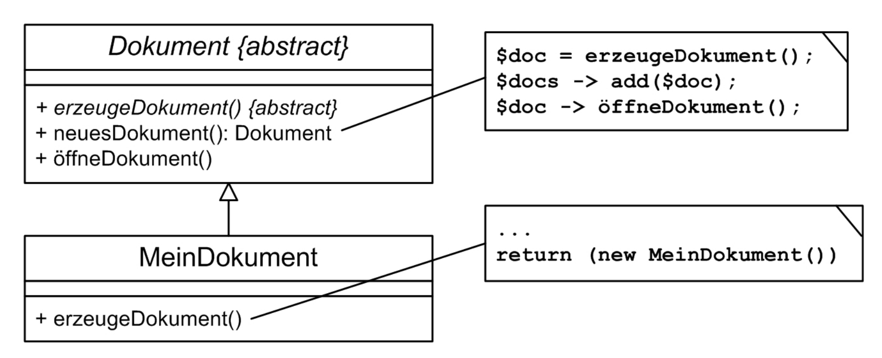


Abbildung 5.40: Skizze des Design Patterns „Fabrik“

Nur ein Exemplar seiner Art: Singleton

Es kann vorkommen, dass Sie eine Klasse programmieren, von der Sie für den Benutzer nur genau ein Objekt anlegen dürfen. So sollten Sie beispielsweise nicht mehrere Datenbankverbindungen gleichzeitig zu einem Datenbankserver aufbauen, da bei einer hohen Anzahl an Benutzer mehrere Verbindungen pro Benutzer den Datenbankserver überlasten können. Werden von mehreren PHP-Skripten Datenbankverbindungen benötigt, so sollten die Skripte stets dieselbe Referenz auf ein einziges Datenzugriffsobjekt besitzen.

Weitere Beispiele für ein Singleton sind Klassen, die Hardwarekomponenten kapseln oder Klassen zur Objektverwaltung, wie eine Kunden- oder eine Aktienverwaltung.

Da ein Objekt keine Kenntnis von der Existenz der anderen Objekte hat, muss die Klasse selbst die Verwaltung übernehmen. Die Lösung besteht darin, eine Klasseneigenschaft *$instance* zu definieren, die eine Referenz auf das einzige Objekt dieser Klasse erstellt. Damit man nicht mehrere Objekte anlegen kann, wird der Konstruktor der Klasse auf *private* gesetzt. Er wird innerhalb der Klasse über eine Klassenmethode aufgerufen, die zunächst prüft, ob *$instance* noch *null* ist. Ist dies der Fall, so wird einmalig der Konstruktor aufgerufen. Andernfalls wird die Instanz auf das bereits bestehende Objekt zurückgegeben.

Singleton
- instance: Singleton
- *__construct()* - getInstance(): Singleton

Abbildung 5.41: Das Design Pattern „Singleton“ als UML-Diagramm

Abbildung 5.41 zeigt das UML-Diagramm eines Singletons, wobei die Bezeichner ebenfalls genormt sind. So können andere Programmierer das Singleton leichter identifizieren. Da es sich dabei um das einfachste Design Pattern handelt, wird in Listing 5.27 eine Implementierung in PHP vorgestellt.

```
<?php
  class Singleton {
    private static $instance = NULL;
    // privater Konstruktor:
    private function __construct() {}
    // die statische Methode gibt die Instanz zurueck
    public static function getInstance() {
      if (self::$instance === NULL) self::$instance = new self;
      return self::$instance;
    }
   // das Klonen von außen muss verboten werden
```

Listing 5.27: Implementierung des Singletons in PHP

```
   private function __clone() {}
  }
  // Aufruf der Instanz
  $singleton = Singleton::getInstance();
?>
```

Listing 5.27: Implementierung des Singletons in PHP (Forts.)

Zwei Schnittstellen zusammenbringen mit einem Adapter

Ihnen ist es bestimmt bereits passiert, dass Sie ein elektronisches Gerät wie einen Fön mit in andere Länder genommen haben und dort an das Stromnetz anschließen wollten. Dort finden Sie jedoch eine andere Steckdose (bzw. Schnittstelle) vor. Damit Sie Ihren Fön betreiben können, benötigen Sie einen Adapter.

Auch eine größere PHP-Anwendung muss mit anderen Anwendungen über Schnittstellen kommunizieren. Dabei kann es auch vorkommen, dass eine Kommunikation zu neuen Schnittstellen erforderlich wird, die bei der Erstellung der PHP-Anwendung noch nicht existierten.

Um den Vergleich mit dem Adapter für den Fön zu wahren, entfernen viele Programmierer die Steckdose und verlegen neue Kabel bis zum Zählerkasten, um ihren Fön anzuschließen. Aber auch in der Softwareentwicklung können Sie einen Adapter bauen.

Der Adapter in der Softwareentwicklung wird eingesetzt, wenn eine existierende Klasse verwendet werden soll, deren Schnittstelle nicht der benötigten Schnittstelle entspricht. Dies tritt insbesondere dann auf, wenn Klassen verwendet werden sollen, die zur Wiederverwendung konzipiert wurden (beispielsweise Werkzeugsammlungen oder Klassenbibliotheken).

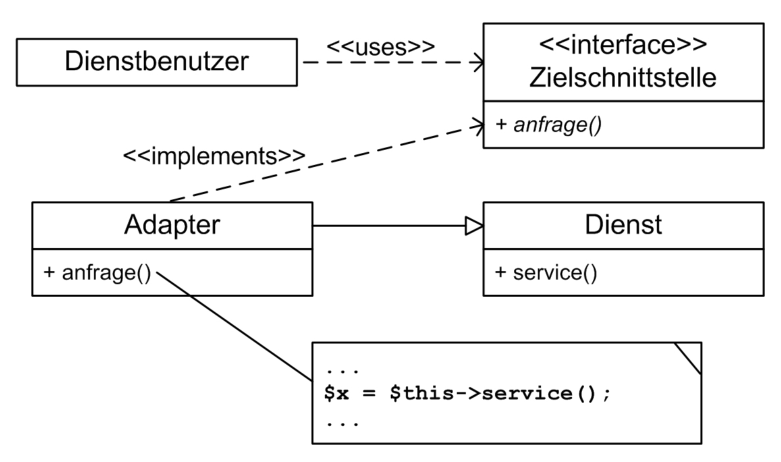


Abbildung 5.42: Das Design Pattern „Adapter" als UML-Diagramm

Diese stellen ihre Dienste durch klar definierte Schnittstellen zur Verfügung, die nicht geändert werden sollen. Dies ist auch häufig nicht möglich, da sie von fremden Herstellern stammen. Des Weiteren wird der Adapter bei der Erstellung wiederverwendbarer

Klassen benutzt, wenn diese mit unabhängigen oder nichtvorhersehbaren Klassen zusammenarbeiten sollen. Abbildung 5.42 zeigt einen Klassenadapter, der einem Dienstbenutzer über eine neue Schnittstelle den Zugriff auf einen vorhandenen Service bietet.

Etwas zusammenbauen mit einem Kompositum

In der Objektorientierung wird ein Ganzes häufig aus anderen Objekten (Teilen) zusammengesetzt. Man spricht hier von einer Aggregation oder einer Komposition. Wie löst man jedoch das Problem, dass ein Teil selbst wieder ein Ganzes ist? Dies klingt auf den ersten Blick verwirrend und vielleicht sogar unrealistisch.

Betrachten Sie jedoch einmal ein Zeichenprogramm. Dort können Sie auf einer Zeichenfläche Dreiecke, Vierecke und Kreise zeichnen. Mit diesen Programmen kann man auch mehrere Elemente gruppieren, um sie dann gemeinsam zu behandeln, zusammen zu verschieben oder mit einer einheitlichen Füllfarbe zu versehen. Die Gruppe selbst ist dann eine Zeichnung in der Zeichnung. Auch mehrere Gruppen können wiederum gruppiert werden. So ergibt sich eine Baumstruktur mit der Zeichenfläche als Wurzel, den Gruppen als Knoten und den Elementen (Dreiecke, Vierecke und Kreise) als Blätter.

Das Design Pattern *Kompositum* setzt Objekte zu Baumstrukturen zusammen, um Teil/Ganzes-Hierarchien darzustellen. Es ermöglicht es, einzelne Teile und das Ganze, also eine Gruppe, gleich zu behandeln.

Abbildung 5.43 skizziert das Kompositum mit einer Methode *zeichnen*, die jedes Element der Gruppe überschreibt, um es auf der Zeichenfläche darzustellen. Wird die Methode *zeichnen* für eine Gruppe aufgerufen, so wird der Aufruf an die Einzelteile der Gruppe weiter delegiert.

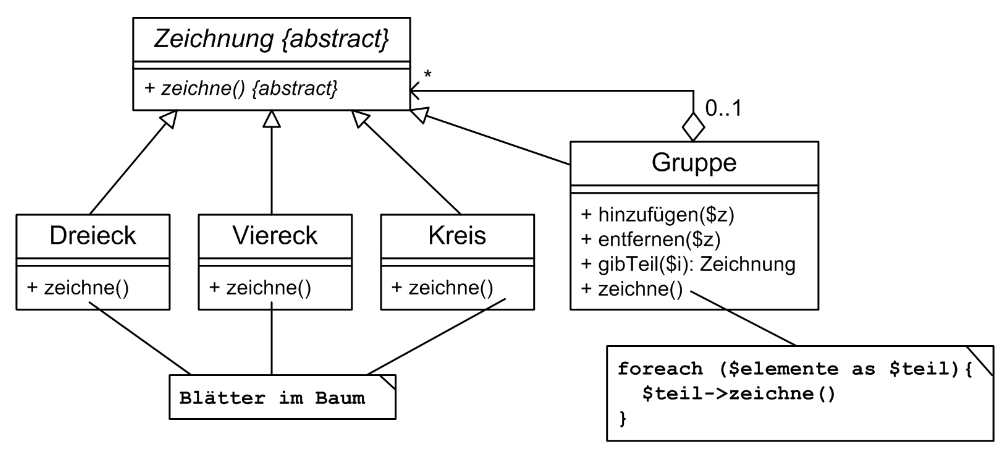


Abbildung 5.43: Das Design Pattern „Kompositum“ als UML-Diagramm

Skizzierung eines variablen Algorithmus: Schablone

Häufig existieren Abläufe, die man prinzipiell formulieren kann und die allgemeingültig sind. Wenn Sie beispielsweise ein neues Dokument öffnen wollen, fügen Sie die Objektreferenz in die Liste der geöffneten Dokumente hinzu, reservieren den Speicher für das

neue Dokument, öffnen eine Verbindung im Dateisystem und lesen das Dokument dann ein. Wie beispielsweise das Einlesen funktioniert, hängt aber vom konkreten Dateityp ab. Eine XML-Datei wird auf eine andere Art und Weise eingelesen als eine Binärdatei.

Das Design Pattern der Schablone definiert in einer Methode den Rumpf eines Algorithmus und delegiert einzelne Teilschritte an Unterklassen weiter. Dies wird angewendet, um invariante Teile eines Algorithmus nur einmal zu kodieren und variierende Teile in konkreten Unterklassen zu implementieren. Die Schablone wird angewendet, wenn ein gemeinsames Verhalten in einer Oberklasse realisiert werden soll, um Codeduplikation zu vermeiden und Erweiterungen der Unterklassen zu kontrollieren, indem die Oberklasse einen allgemeingültigen Ablauf vorschreibt. Ein allgemeines Schema zur Umsetzung einer Schablone ist in Abbildung 5.44 skizziert.

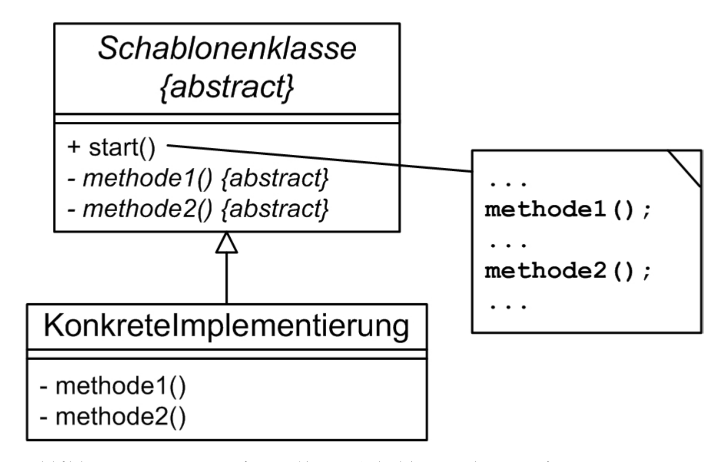


Abbildung 5.44: Das Design Pattern „Schablone“ als UML-Diagramm

Viele beobachten und ändern eine Datenquelle: Beobachter

Das letzte vorgestellte Design Pattern wird in der Kategorie der objektbasierten Verhaltensmuster eingeordnet. Nehmen wir an, Sie haben einen Datenstamm in Form einer Tabelle, der auf verschiedene Art und Weise aufbereitet werden soll, beispielsweise als Linien-, Balken oder Kuchendiagramm. Alle Darstellungen befinden sich auf einer einzigen Seite. Nehmen wir weiter an, dass in jeder Darstellung der Datenstamm verändert werden kann, zum Beispiel soll durch ein Ziehen an einem Balken nach oben der Wert dieses Datensatzes vergrößert werden. In diesem Fall soll dadurch die Datenquelle aktualisiert werden, was wiederum eine Aktualisierung aller Darstellungen zur Folge hat. Abbildung 5.45 skizziert den Fall.

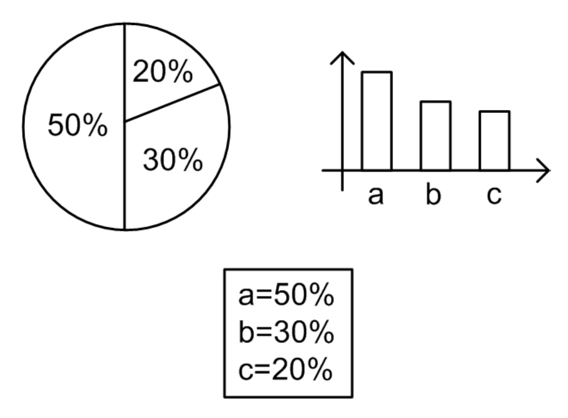


Abbildung 5.45: Anwendungsfall für ein Kompositum

Nun soll diese Aktualisierung natürlich ohne Verzögerung erfolgen, sobald der Anwender eine Darstellung verändert hat. Zusätzlich dazu ist die Anzahl der verschiedenen Darstellungen nicht im Vorfeld bekannt und es können neue, noch unbekannte Visualisierungen des Datenbestands hinzukommen. Wenn Sie dies manuell realisieren wollen, ist der Programmieraufwand sehr hoch. Ebenfalls ist es wahrscheinlich, dass Sie zunächst Probleme mit dem Update des Datenbestands bekommen, mit der Geschwindigkeit der Aktualisierung und ggf. sogar eine Endlosschleife erzeugen, falls die Aktualisierungen versehentlich zu einer Rekursion führen. Als fertige Lösungsstrategie für dieses Problem dient das Kompositum.

Ein Kompositum sorgt dafür, dass bei Änderung eines Objekts alle davon abhängigen Objekte benachrichtigt werden. Diese können dann ihren Zustand aktualisieren. Die Objekte sind also wechselseitig voneinander abhängig, jedoch ist es unbekannt, wie viele Objekte geändert werden müssen. Die Datenquelle soll bei einer Änderung die anderen Objekte benachrichtigen, sodass eine lose Kopplung der Objekte untereinander entsteht.

Um dies umzusetzen, besitzt ein Objekt der Klasse *Datenquelle* einen exklusiven Zugriff auf einen Datenbestand. Alle Objekte, die sich für die Daten interessieren, müssen sich bei dem Datenquellenobjekt anmelden. Diese werden Beobachter genannt. Somit besitzt das Datenquellenobjekt eine Liste aller Beobachter, die sich für den Datenbestand interessieren. Das Datenquellenobjekt benachrichtigt dann bei einer Änderung alle Beobachter, die daraufhin ihren Zustand aktualisieren.

Die notwendigen Methodenaufrufe bei einer solchen Aktualisierung sind in dem Sequenz-Diagramm der Abbildung 5.46 dargestellt. Über die Methode *update* informiert das Datenquellenobjekt nacheinander alle Beobachter. Die Beobachter fordern dann mit der Methode *getState* nur die Daten von dem Datenquellenobjekt an, die sie interessieren.

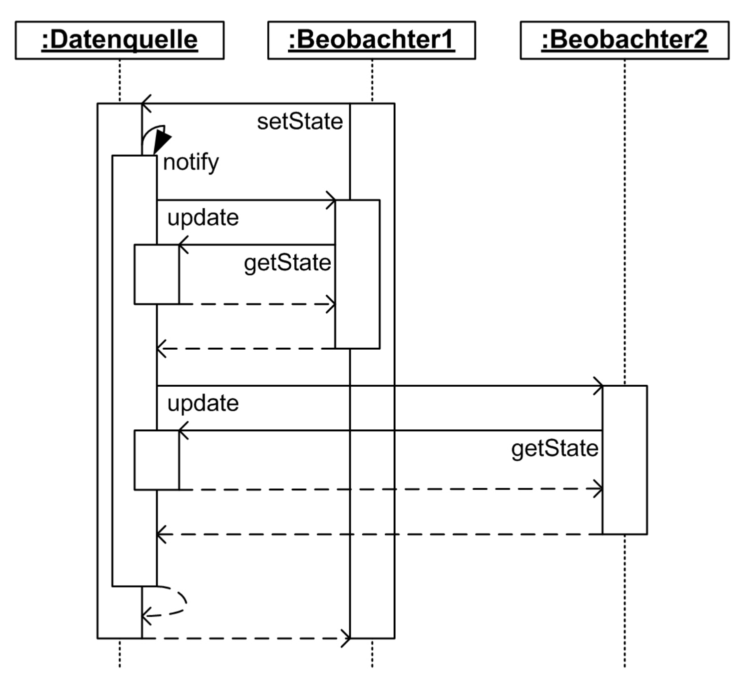


Abbildung 5.46: Das Design Pattern „Kompositum" als Sequenzdiagramm

5.2.3 Wie man es nicht machen sollte – Anti-Pattern

Als Gegenbewegung zu den Design Patterns wurden in den letzten Jahren auch typische schlechte Lösungsansätze festgehalten, die mit der Entwicklung und Einführung einer großen, objektorientierten Anwendung häufig vorzufinden sind. Diese schlechten Lösungsansätze bezeichnet man als Anti-Pattern.

Die Kenntnis von typischen Anti-Pattern trägt dazu bei, wiederkehrende Fehler bei der Softwareentwicklung zu identifizieren, zu dokumentieren und Maßnahmen aufzuzeigen, wie sie behoben werden können. Die richtige Anwendung von Design Patterns kann dazu beitragen. In der Regel entstehen Anti-Pattern wider besseres Wissen oder durch mangelnde Erfahrung im Verlauf der (objektorientierten) Softwareentwicklung. Da sich die Anti-Pattern nicht nur auf die Programmierung beschränken, wurden typische Fehler klassifiziert in

- Anti-Pattern der Programmierung
- Anti-Pattern der Systemarchitektur
- Anti-Pattern des Projektmanagements
- Anti-Pattern des Managements

Die bekanntesten Vertreter aus diesen Kategorien werden im Folgenden kurz vorgestellt. Auch wenn die Darstellung teilweise übertrieben ist, können Sie wahrscheinlich Muster erkennen, die Ihnen in Ihrem Alltag als Programmierer, Systemanalytiker oder (freiberuflicher) Projektleiter bereits begegnet sind.

Copy/Paste-Programmierung

Da viele Klassen eine ähnliche Struktur, insbesondere bei den Get- und Set-Methoden besitzen, neigen viele Entwickler dazu, bestehenden Quellcode zu kopieren. Dies gilt auch für bestehende Algorithmen. Dabei besteht jedoch die Gefahr, Fehler mitzukopieren oder dass die Kopie für den neuen Bereich nicht optimal einsetzbar ist. Oft denkt der Entwickler nicht mehr über den genauen Sinn des Quellcodes nach.

Werden gleichförmige Strukturen in mehreren Abläufen erkannt, so könnte das Schablonen-Design-Pattern der Copy/Paste-Programmierung vorbeugen und doppelten, schlecht wartbaren Quellcode vorbeugen.

Spaghetticode

Dabei handelt es sich um eine minimale Systemarchitektur, deren Kontrollfluss einem Topf Spaghetti ähnelt. Die Gefahr besteht oft in Verbindung mit Rapid Prototyping, wenn Phasen des Refactorings bzw. ordentliche Klassenmodellierungen ausbleiben.

Zwiebel-Programmierung

Ein wichtiges Ziel objektorientierter Programmierung ist die Wiederverwendbarkeit. Dabei wird neue Funktionalität über die alte Funktionalität gelegt. Dies ist häufig zu beobachten, wenn ein Entwickler ein Programm erweitern soll, welches er nicht geschrieben hat. Er setzt seine neue Lösung einfach darüber.

Dies führt mit einer Vielzahl von Versionen und unterschiedlichen Entwicklern über die Jahre zu einem Zwiebelsystem. Unterbinden werden kann dies durch strukturierte und gut dokumentierte Klassenbibliotheken, die auch konsequent eingesetzt werden sollen. Dabei ist vom Management zu unterbinden, dass vorhandene Funktionen aus einer Klassenbibliothek nicht noch einmal neu geschrieben werden sollen.

Lavafluss

Wenn ein Quellcode mehrere Versionen durchläuft, kommt es vor, dass immer mehr „toter Quellcode“ herumliegt, der beispielsweise zu Debug-Zwecken verwendet wurde. Andere Entwickler können nicht nachvollziehen, ob dieser Quellcode noch eine Funktion hat oder nicht. Da es ja funktionierte, bauen sie ihre Erweiterungen um diesen Quellcode herum, statt ihn zu entfernen nach dem Motto „never touch a running system“. Konsequentes Refactoring ist eine Maßnahme gegen den Lavafluss.

Programmierung mit dem Switch-Statement

Der Zustand eines Objekts wird durch die aktuellen Ausprägungen seiner Eigenschaften gesteuert. Viele Entwickler benutzen eine zusätzliche Statuseigenschaft, die den aktuellen Zustand des Objekts repräsentiert und die über eine Get-Methode abgefragt werden kann.

In den Set-Methoden wird dann geprüft, ob das Setzen einer Eigenschaft gemäß dem aktuellen Zustand erlaubt ist. Dies geschieht zumeist über die Verwendung des switch-Statements oder einer mehrfachen Verzweigung.

Den Zustand sollte man jedoch nicht mit hart kodiertem Quellcode verwalten. Stattdessen sollte der Zustand eines Objekts über das Status-Design-Pattern mithilfe von eigenen Statusobjekten verwaltet werden.

Das quadratische Rad neu erfinden

Objektorientierte Programmiersprachen bieten bereits eine große Anzahl an vorgefertigten Klassenbibliotheken. Weitere Funktionalität kann immer stärker über frei erhältliche Open-Source-Lösungen bezogen werden, bei denen eine Gemeinschaft über die Entwicklung und die Beseitigung von Fehlern wacht. Dies geht hin bis zu komplexen Frameworks, die ganze Lösungen bereitstellen.

Viele Programmierer (insbesondere aus der prozeduralen Welt) glauben jedoch, dass Sie diese Funktionalität „mal eben schnell selbst" implementieren können. Schwierigkeiten sind dabei in der Regel nicht zu Beginn erkennbar, die andere bereits gelöst haben.

Dies führt dazu, dass versucht wird, das Rad nochmals neu zu erfinden. Das Ergebnis ist meist ein Rad, das nicht so rund läuft wie es sollte. Der Aufwand für die Entwicklung war wesentlich höher als zunächst angenommen. Recherchieren Sie also zunächst nach kostenfreien bzw. kostengünstigen Alternativen, bevor Sie eine Neuentwicklung beginnen. Meist hat bereits jemand anders vor Ihnen dieses Problem gehabt und es auch gelöst. Und selbst wenn Sie sich für eine Neuentwicklung entscheiden, sollten Sie zumindest die existierenden Lösungsansätze kennen.

Die Wunderwaffe

Im letzten Kapitel haben Sie einige Design Patterns gesehen, von denen Sie vielleicht bereits eines implementiert haben. Wenn man ein Design Pattern oder generell eine Vorgehensweise versteht, neigt man dazu, es überall anzuwenden. Die Problemstellung wird dann auf das Muster bzw. auf die bekannte Vorgehensweise angepasst, obwohl dies eigentlich nicht möglich ist.

Die bekannte Vorgehensweise wird somit als Wunderwaffe angesehen, die jedes Problem löst. Entgegenwirken kann man der Wunderwaffe durch ein breites Spektrum an Kenntnissen, Offenheit für Neues und durch Dialog mit allen Beteiligten.

Eine Systemarchitektur als außerirdische Spinne

Zu Beginn eines objektorientierten Designs können viele Analytiker eine Trennung der einzelnen Klassen nur schwer vornehmen. Sie sind der Meinung, dass jedes Objekt ja irgendwie jedes andere kennen muss.

Das Resultat sind sehr kommunikative Objekte, die sich alle gegenseitig kennen. Es können keine Komponenten aus der Anwendung extrahiert und wiederverwendet werden. Alles gehört irgendwie zusammen und ist von allem anderen abhängig. Die verwobene Struktur wird als außerirdische Spinne bezeichnet, der man am Besten mit einem konsequenten modularen Design unter Verwendung von Design Patterns begegnet.

Eine Systemarchitektur mit Gottklasse

Eine ähnliche Problemstellung ergibt sich, wenn Sie davon überzeugt sind, eine zentrale Managerklasse zu benötigen. Ein Objekt dieser Klasse weitet sich schnell zu einem Gottobjekt aus, das alle Aufgaben übernimmt, alle anderen Objekte kennt und verwaltet. Andere Klassen existieren oft nicht oder haben lediglich die Funktion von Datencontainern für die Gottklasse.

Eine Aufteilung nach Verantwortlichkeiten durch Bildung von Unterklassen, Datenkapselung und die Verwendung von Design Patterns sind dabei behilflich, die Gottklasse zu verhindern.

Eine Systemarchitektur als innere Plattform

Um auf zukünftige Erweiterungen vorbereitet zu sein, versuchen viele Analytiker und auch Entwickler, alle möglichen zukünftigen Erweiterungen im Vorfeld zu berücksichtigen. Dadurch entsteht eine Anwendung mit derartig weitreichenden Konfigurations-Möglichkeiten, dass die eigentliche Funktionalität im Quellcode nicht mehr erkennbar ist.

Ein Beispiel dafür sind quasiflexible Datenmodelle, die auf anwendungsbezogene Datenbanktabellen wie *Kunde*, *Artikel* und *Rechnung* verzichten und stattdessen mittels allgemeiner Tabellen eine eigene Verwaltungsschicht für die Datenstruktur implementieren wollen.

Um dem Plattformeffekt zu begegnen, müssen Sie sich zunächst bewusst sein, dass Sie niemals alle Anforderungen und Erweiterungen im Vorfeld erkennen und berücksichtigen können. Oft entstehen neue, unvorhersagbare Anforderungen, die auch Ihrem Kunden nicht beim Start des Projekts bekannt sein konnten.

Die Verwendung agiler Methoden mit einer ausgiebigen Kommunikation aller Beteiligten sowie ein robustes Design der Anwendung mit gekapselten, wiederverwendbaren Komponenten helfen zusätzlich, den Plattformeffekt zu vermeiden und dennoch eine erweiterbare Anwendung zu erzeugen.

Sumo-Hochzeit

Viele Datenbankhersteller bieten an, einen Teil der Fachlogik in gespeicherten Prozeduren innerhalb des Datenbankservers zu realisieren. Außerdem wird der standardisierte SQL-Befehlssatz oft erweitert. Dies hat zur Folge, dass der Client unnatürlich stark von der Datenbank abhängig ist. Auch wenn dies einen Gewinn an Performance bietet, wird die Architektur der Anwendung dadurch unflexibel und es entsteht eine Abhängigkeit vom Datenbankhersteller.

Wird die Anwendung zu einer Internetanwendung migriert oder die Datenbank ausgetauscht, so müssen auf beiden Schichten viele Bereiche neu entwickelt werden. Begegnen kann man der Sumo-Hochzeit mit einer klaren Trennung der Schichten in eine Datenzugriffsschicht, die alle SQL-Anweisungen kapselt, und konsequenter Verwendung von Standardbefehlen der SQL, einer separaten Fachlogik und einer ebenfalls separaten Schicht für die Benutzerinteraktion.

Blendwerk des Projektmanagements

Eine Gefahr bei GUI-Prototypen besteht darin, dass nicht fertige Funktionen als fertig vorgetäuscht werden. Wenn Ihr Kunde sich bereits durch die gesamte Anwendung klicken kann, so kann er die Frage stellen, wozu die restlichen 70 % der Gelder notwendig sind, die noch bezahlt werden sollen.

Erfahrungsgemäß haben Anwender gewöhnlich nur wenig Kenntnis von dem Backend hinter der Benutzeroberfläche. Gegen das Blendwerk hilft es, neben horizontalen GUI-Prototypen auch in einer frühen Phase des Projekts vertikale Prototypen zu implementieren, die eine Funktion durchgängig durch alle Schichten implementiert. An dieser Entwicklung sollte der Kunde beteiligt sein, um gemeinsam eine genauere Aufwandsabschätzung für jede zu realisierende Funktionalität vornehmen zu können.

Erschleichung von Funktionalität

Der Umfang der zu entwickelnden Funktionalität wird gewöhnlich in einem Projektplan festgehalten. Die Erschleichung von Funktionalität besteht dann, wenn Ihr Kunde nach der Erstellung des Projektplans versucht, weitere Funktionalität in der Version mit unterzubringen. Dabei wird oft angemerkt, dass einige Funktionen doch selbstverständlich sind. Ohne diese Funktionen sei die zu erstellende Anwendung nicht funktionsfähig.

Dies ist dann problematisch, wenn die aktuelle Iteration nicht das notwendige Design aufweist, Termine nicht eingehalten werden können oder die Kosten explodieren.

Begegnen kann man der Erschleichung von Funktionalität durch einen offenen und ehrlichen Umgang aller Beteiligten an dem gemeinsamen Projekt. Gleichzeitig sollten die Ergebnisse von Meetings und die erstellten Angaben zu Spezifikation stets von allen Beteiligten festgehalten und unterzeichnet werden. Fragen Sie als Analytiker oft nach, wenn Ihnen ein beschriebener Geschäftsprozess, der implementiert werden soll, unvollständig erscheint. Eine iterativ-inkrementelle Entwicklung mit Teilrechnungen ist auch oft für alle Beteiligten ein effizientes Mittel, um den Überblick zu behalten.

Death Sprint

Der Death Sprint entsteht durch die übereifrige Erfüllung der nächsten Iteration. Die Prototypen werden iterativ bereitgestellt, jedoch in zu kurzen Zeitintervallen. Aus Sicht des Kunden und sogar aus Sicht Ihres eigenen Managements sieht das Projekt zunächst erfolgreich aus, da immer wieder neue Versionen mit neuen Eigenschaften abgeschlossen werden.

Dabei leidet jedoch die Softwarequalität, was zunächst nur von den Entwicklern wahrgenommen wird. Das Design der Anwendung kann nicht mit den Iterationen mithalten.

Um einen Death Sprint zu vermeiden, sollten bereits zu Beginn des Projekts in jede Iteration eine Phase des Testens sowie ein Refactoring eingeplant werden, um die neu entwickelten Funktionen sauber in das bestehende Systemdesign zu integrieren.

5.2.4 Entwicklungsumgebungen und Tools

Im letzten Kapitel dieses Buches werden Werkzeuge und Entwicklungsumgebungen vorgestellt, die Ihnen bei der Entwicklung von PHP-Projekten behilflich sein sollen.

Klassifizierung von UML-Werkzeugen zur Entscheidungshilfe

Noch bevor Sie Tests entwerfen, Quellcode versionieren oder dokumentieren, stehen die objektorientierte Analyse und das objektorientierte Design mit UML. Zu diesem Zweck existiert eine Vielzahl von UML-Werkzeugen, die auf verschiedene Arten klassifiziert werden können.

Die erste Entscheidung besteht darin, ob Sie aus Ihren UML-Diagrammen Quellcode erzeugen wollen oder nicht. Einige Tools sind lediglich Zeichenprogramme, mit denen Sie zu Zwecken der Diskussion und Dokumentation UML-Diagramme erstellen. Diese Tools sind meist leicht zu handhaben, jedoch müssen Sie einen gewissen Aufwand einplanen, um UML-Diagramme und Quellcode auf einer einheitlichen Version zu halten.

Wenn Sie sich für ein quellcodegenerierendes Werkzeug entscheiden, so können Sie zumeist aus Klassendiagrammen direkt PHP-Coderümpfe erstellen. Einige Werkzeuge unterstützen bereits die Umsetzung von Zustands- und Aktivitätsdiagrammen in PHP-Code. Hierbei ist entscheidend, dass das Werkzeug auch PHP-Code generieren kann. Die meisten Tools sind dabei auf Java spezialisiert, einige erzeugen ausschließlich C#- oder VB.NET-Quellcode. Zu prüfen ist auch die Frage, wie gut Sie mit generiertem PHP-Code umgehen können. Viele Werkzeuge erzeugen kryptischen Code, den ein Entwickler im Nachhinein nur schwer lesen kann.

Eine weitere Entscheidung innerhalb der quellcodegenerierenden Werkzeuge besteht darin, ob sie Re-Engineering unterstützen oder nicht. Im ersten Fall können Sie modifizierte PHP-Skripte in das Tool zurückladen und die UML-Darstellung automatisch aktualisieren. Im zweiten Fall modellieren Sie eher und erzeugen stets neuen PHP-Code. Einige Tools überschreiben dabei durchgeführte Änderungen am Quellcode.

Weitere Unterscheidungskriterien in den UML-Werkzeugen liegen darin, welche UML-Diagrammarten von dem Werkzeug unterstützt werden. Mit reinen Zeichenprogrammen können Sie natürlich beliebige Diagramme erstellen, die jedoch nicht unbedingt dem UML-Standard entsprechen. UML-Werkzeuge schränken Sie hier zugunsten des Standards ein. Bei nahezu allen Werkzeugen werden Klassendiagramme unterstützt. Nur einige davon unterstützen zusätzlich die Erstellung von Anwendungsfall-, Aktivitäts-, Sequenz- und Zustandsdiagrammen.

Letztlich ist noch als Unterscheidungskriterium zu nennen, ob Sie ein Open-Source-Werkzeug oder ein lizenzpflichtiges Tool einsetzen wollen. Die erste Version ist natürlich kostengünstiger und bei einigen Tools erfolgt eine Unterstützung des Werkzeugs durch eine entsprechend große Community. Die Lebensfähigkeit des Tools ist andererseits genauso von der Community abhängig.

Bei einem lizenzpflichtigen Tool bezahlen Sie hingegen zunächst einmal Geld, kaufen jedoch meist einen Support des Herstellers mit ein. Lizenzpflichtige Tools können Sie meist zunächst in einer kostenfreien Testversion mit beschränkter Laufzeit, aber (nahezu) vollem Funktionsumfang erwerben. Generell sollten Sie sich zunächst mit eini-

gen Werkzeugen vertraut machen, um sich mit dem Umgang anzufreunden. Letztlich sollten Sie mögliche Lizenzkosten gegen einen eventuellen Mehraufwand in Mannstunden bei der Verwendung eines anderen Tools abwägen. Gerade bei großen Projekten sollte der effiziente Umgang mit den Werkzeugen absolut im Vordergrund stehen.

Eine aktuelle Liste von UML-Werkzeugen finden Sie zusammen mit den Links zu den Herstellern bzw. zu den Downloads auf der englischsprachigen Wikipedia-Seite *http://en.wikipedia.org/wiki/List_of_UML_tools*. Hervorzuheben sind aus dieser Liste insbesondere

- ArgoUML für Java-unterstützende Betriebssysteme (*http://argouml.tigris.org/*)
- Visual Paradigm for UML für Java-unterstützende Betriebssysteme (*http://www.visual-paradigm.com/product/vpuml/*)
- Umbrello UML Modeller (*http://uml.sourceforge.net/*) für Linux
- WinA&D für Windows (*http://www.excelsoftware.com/wina&dproducts.html*)

aufgrund ihrer Bekanntheit und ihrer Unterstützung von PHP-Quellcode.

UML-Diagramme mit Microsoft Visio erstellen

Als exemplarisches UML-Werkzeug, das auch bei der Erstellung dieses Buches eingesetzt wurde, wird Microsoft Visio 2003 Professional vorgestellt. Die Bedienbarkeit der neuen Version, Microsoft Visio 2007, ist nahezu identisch, da Microsoft bei Visio 2007 nicht das neue Konzept der Menüführung mit zentraler Office-Schaltfläche oben links im Fenster etabliert hat. Das neue, in Microsoft Word, Excel und Access 2007 eingesetzte Bedienkonzept ist größtenteils auf Kritik der Anwender gestoßen.

Im Gegensatz zu der Standardversion besitzt die Professional-Version eine eigene Sammlung von UML-Schablonen, von deren Einsatz jedoch abzuraten ist. Die Ursache liegt darin, dass die Handhabung unpraktischer ist und eine PHP-Quellcodegenerierung nicht unterstützt wird. Stattdessen werden eigene UML-Symbole gezeichnet und verwendet. Somit kommen wir zu der Klassifizierung dieses Werkzeugs:

- Es handelt sich bei Microsoft Visio um ein reines Zeichenwerkzeug, mit dem kein PHP-Quellcode erzeugt werden kann. Die erstellten Diagramme können zur Diskussion und Dokumentation vor allem in der OOA und OOD verwendet werden.
- Dadurch, dass Visio ein Zeichenwerkzeug ist, mit dem Linien, Pfeile, Kreise/Ellipsen und Vierecke gezeichnet und mit Text versehen werden, können alle UML-Diagramme in ihren Notationen umgesetzt werden.
- Es handelt sich um eine kostenpflichtige Lizenz, die sich auf einen Arbeitsplatz bezieht. Unternehmen, die ohnehin einen Lizenzvertrag mit Microsoft besitzen (Stichwort: MSDN – Microsoft Developer Network), können in der Regel vergünstigt auf eine Visio-Lizenz zugreifen. Andererseits widerspricht der Open-Source-Gedanke von PHP und beispielsweise von dem XAMPP-Paket dem lizenzpflichtigen Modell.

Für Visio spricht jedoch die einfache Handhabung eines komplexen vektororientierten Zeichenprogramms aus Anwendersicht sowie die Integration in das weit verbreitete Office-Paket von Microsoft.

Eine Visio-Zeichnung (VSD – Visio Drawing) besteht aus mehreren Zeichenblättern, die eine nahezu beliebige Größe einnehmen können. Die Anwendung verfügt über alle notwendigen Zeichenfunktionen sowie die Möglichkeit, eigene Symbolbibliotheken anzulegen (VSS – Visio ShapeSheet). Die erstellten Zeichnungen können in Pixelformate (JPG, GIF) exportiert oder auch direkt über OLE (Object Linking and Embedding) mit anderen Office-Anwendungen ausgetauscht werden.

Eine Vollversion von Microsoft Visio 2007 Standard kostet derzeit ca. 329 €, ein Upgrade ca. 159 €.

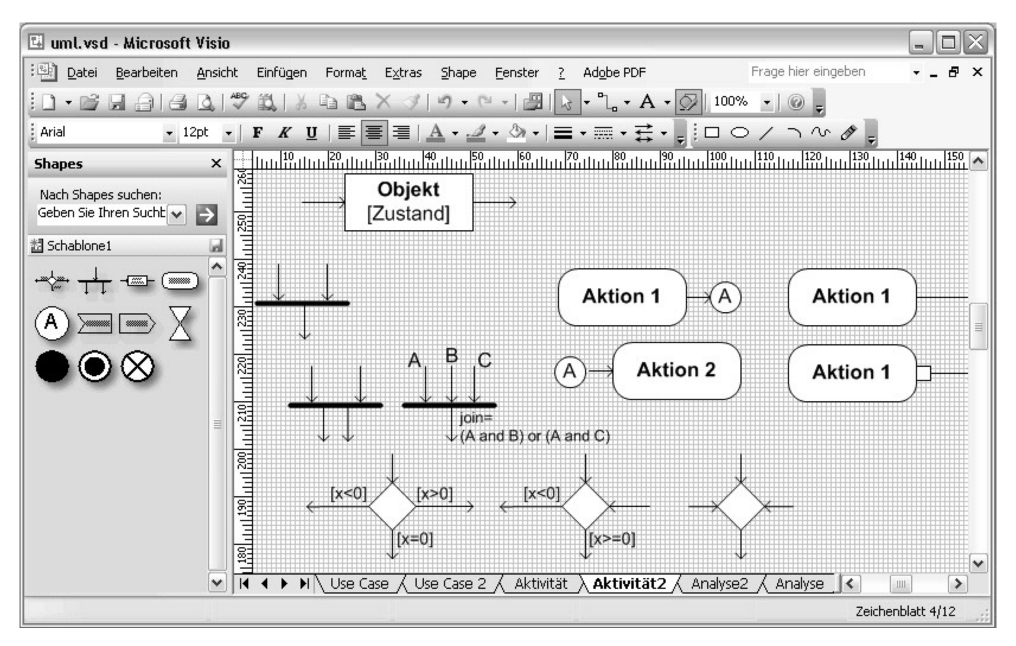


Abbildung 5.47: Screenshot von Visio 2003

Automatisch testen mit PHPUnit

Das aus der Java-Welt bekannte Werkzeug JUnit ist nun auch für PHP unter dem Namen PHPUnit verfügbar. Es ist bei der Umsetzung der agilen testgetriebenen Entwicklung behilflich, indem noch vor der Implementierung einer Funktionalität separate Testklassen geschrieben werden. Diese Testklassen ersetzen die Testskripte, wie sie in Kapitel 5.1, beispielsweise in Listing 5.3 verwendet wurden. Eine Testklasse besteht aus beliebig vielen Testmethoden, die alle hintereinander geschrieben und bei jedem Testdurchlauf ausgeführt werden. So sammelt sich eine Vielzahl von Tests. Jeder erfolgreiche Testdurchlauf dient dem Nachweis der Robustheit der Software und dokumentiert gleichzeitig den Kontext, in dem die zu testende Klasse eingesetzt wird.

Nach der Installation über den bereits in PHP integrierten PEAR-Installer (PHP Extension and Application Repository), der die Verwaltung von PHP-Zusatzmodulen übernimmt, steht PHPUnit als Kommandozeilenwerkzeug zur Verfügung. Dadurch kann es in Batch-Routinen eingebunden werden und somit ein automatisiertes Testen ermöglichen, bevor ein neues Release der PHP-Anwendung erstellt wird.

Das Schreiben der Tests erfolgt durch Einbinden des Frameworks sowie durch die Erstellung einer Testklasse, die von *PHPUnit_Framework_TestCase* abgeleitet wird. Über die Vererbung steht dem Tester nun eine Reihe von Assert-Methoden zur Verfügung, mit denen er Soll- und Ist-Ausgaben vergleichen kann. Entspricht eine Soll-Ausgabe nicht der entsprechenden Ist-Ausgabe, so ist der Test fehlgeschlagen.

```
<?php
  require_once 'PHPUnit/Framework.php';
  class StackTest extends PHPUnit_Framework_TestCase{
    public function test01(){
    $metro=new Aktie("METRO AG Stammaktien o.N.","DE0007257503",
                          "http://www.boerse-....ISIN=DE0007257503");
    $this->assertEquals("METRO AG Stammaktien o.N.", $metro->getName());
    }
  }
?>
```

Listing 5.28: Der Test aus Listing 5.1 als PHPUnit Test

Die wichtigsten Prüfmethoden lauten

- *assertTrue*
- *assertFalse*
- *assertNull*
- *assertSame* zur Prüfung auf gleiche Objektreferenzen
- *assertNotSame*
- *assertEquals* zur Prüfung auf gleiche Inhalte
- *assertNotEquals*
- *assertContains* zur Prüfung, ob ein Element zu einem Feld gehört
- *assertNotContains*
- *assertRegExp* zur Prüfung, ob eine Zeichenkette einem regulären Ausdruck entspricht
- *assertNotRegExp*
- *assertType* zur Prüfung auf Variablentypen
- *assertNotType*

Zusätzlich können Sie selbst in Verzweigungen Prüfungen vornehmen, indem Sie für ungültige Pfade die Methode *fail($string)* aufrufen, um den Test als fehlerhaft zu kennzeichnen.

Die Ausgabe der Konsolenanwendung ist in Abbildung 5.48 dargestellt. Dabei werden zwei Tests durchgeführt, die als Resultat *.F* erzeugen. Der Punkt für den ersten Test zeigt den Erfolg, das F deutet auf das Fehlschlagen des zweiten Tests. Zusätzlich wird die Dauer gemessen, die die Tests für ihre Ausführung benötigen.

```
parth-patils-computer:~/test parth$ phpunit ArrayTest
PHPUnit 3.2.18 by Sebastian Bergmann.

.F

Time: 0 seconds

There was 1 failure:

1) testArrayContainsAnElement(ArrayTest)
Failed asserting that <integer:1> matches expected value <integer:2>.
/Users/parth/test/ArrayTest.php:23

FAILURES!
Tests: 2, Failures: 1.
```

Abbildung 5.48: Konsolenausgabe von PHPUnit

Weitere Informationen zu PHPUnit finden Sie unter *http://www.phpunit.de/*.

Funktionsfähige Skripte verwalten mit SVN-Versionierung

Gerade bei größeren Projekten mit mehreren Entwicklern existiert häufig eine Vielzahl von PHP-Dateien. Einige Entwickler editieren gerade einige dieser Dateien und fügen neue Funktionalität hinzu. Zwischendurch müssen funktionsfähige Prototypen für Kunden erstellt werden. Bei einem Produkt wird die Entwicklung auch nach der Auslieferung der PHP-Anwendung fortgesetzt.

Ein bekanntes Werkzeug unter freier Lizenz, das eine Versionierung der Dateien unterstützt, ist Subversion (SVN) der Firma CollabNet. SVN wird als Nachfolger des weit verbreiteten Concurrent Versions System (CVS) angesehen. Die Versionierung ist dabei unabhängig von der Programmiersprache PHP. Es können beliebige Quellcodedateien und auch Dokumentationen für die Versionierung verwendet werden.

Für die Versionierung wird ein Server im Intranet benötigt, der das PHP-Projekt enthält. Die Versionierung erfolgt in diesem zentralen Projektarchiv in Form einer einfachen Revisionszählung. Das Archiv wird auch als Repository bezeichnet. Wenn ein Entwickler eine Quellcodedatei ändern will, checkt er diese Datei aus und bearbeitet sie. Dies ist jedoch meist nur im Kontext des Gesamtprojekts möglich, da die PHP-Datei ja mit anderen PHP-Dateien verknüpft ist. Daher checken mehrere Entwickler das gesamte Projekt aus und editieren (idealerweise) verschiedene Quellcode-Dateien.

Sobald ein Entwickler fertig ist mit der Bearbeitung seiner Dateien, werden zunächst die PHPUnit-Tests durchgeführt. Im Anschluss daran checkt der Entwickler die geänderten

Dateien über das SVN-System wieder ein. Dabei werden nur die geänderten Dateien auf den Server übertragen. Diese Dateien erhalten dann eine neue Revisionsnummer, die alten Revisionen bleiben stets archiviert, sodass das System gleichzeitig als Sicherungssystem gilt. Sie können also stets eine ältere Revision einer Datei einsehen und auch verändern.

Wenn Sie einen neuen Prototyp anlegen, bedeutet dies, dass Sie von jeder PHP-Datei eine bestimmte Revision verwenden wollen. Die Revisionen bilden in ihrer Gesamtheit dann den Prototyp bzw. das Release, das dem Kunden vorgeführt und auch ausgeliefert werden kann.

Ein Problem entsteht, wenn zwei Entwickler Änderungen an derselben Datei vornehmen. Checkt der erste Entwickler wieder ein, so geschieht dies problemlos. Der zweite Entwickler bekommt jedoch eine Meldung, dass sich eine andere Version der Datei auf dem Server befindet als die, die man ausgecheckt hat. Nun ist zu entscheiden, welche der beiden Versionen verwendet wird oder ob sogar eine neue Version erstellt wird, die alle Änderungen umfasst. Dieses Prinzip nennt man Zusammenführung bzw. Merging. Als Hilfsmittel dazu zählt das in SVN integrierte Hilfswerkzeug *diff*, mit dem man Differenzen zwischen zwei Quellcodedateien farblich hervorheben kann. Dies dient als Entscheidungshilfe für das weitere Vorgehen.

Die Herstellerhomepage zu SVN finden Sie unter *http://subversion.tigris.org/*. Dort können Sie auch die aktuelle Version von Subversion herunterladen. Ein deutschsprachiges Onlinebuch zum Einstieg in die Versionsverwaltung finden Sie unter *http://svnbook.red-bean.com/nightly/de/index.html*.

Dokumentieren mit PHPDocumentator

In der Java-Welt hat sich seit einigen Jahren die Erstellung von Quellcodekommentaren unter Verwendung der Java-Doc etabliert. Die Idee besteht darin, Quellcodekommentare in einem bestimmten Stil so zu erzeugen, dass sie selbst automatisch interpretiert werden können. Die Zeiten, in denen separate Quellcodedokumentationen leidvoll in einem Textverarbeitungsprogramm erstellt wurden, sind also vorüber.

Das über Kommandozeile oder auch über Webinterface bedienbare Werkzeug PHPDocumentator ist auf der englischsprachigen Internetseite *http://www.phpdoc.org/* kostenfrei erhältlich und kann ebenso wie PHPUnit unter Verwendung des PEAR-Installers installiert werden. Neben einem Handbuch finden Sie auf dieser Homepage auch umfangreiche Tutorials zur Verwendung des PHPDocumentators.

Hinweis

Eine Quellcodedokumentation ist selbstverständlich nicht die einzige Dokumentation, die zu erstellen ist. Die Aufgabe, ggf. ein Benutzerhandbuch für die Interaktion von Anwendern mit Ihrer PHP-Anwendung zu erstellen, bleibt Ihnen natürlich nicht erspart.

Um Kommentare interpretieren zu können, ist eine entsprechende Metasprache erforderlich, die bei PHPDocumentator eng an die Java-Doc angelehnt ist. Listing 5.29 zeigt beispielhaft die Kommentierung einer selbstgeschriebenen PHP-Funktion *foo*.

```
/**
  * foo: liefert $bar*$bar zurück.
  *
  * @author     Dr. Frank Dopatka
  * @version    1.0
  * @param      $bar     Ein beliebiger Wert
  * @return     Der mit sich selbst multiplizierte Eingabewert
  */
  function foo($bar) {
    return -$bar;
  }
```

Listing 5.29: Eine eigene PHP-Funktion mit Kommentierung für PHPDocumentators

Die Metabefehle werden also stets durch ein vorangestelltes @-Zeichen eingeleitet. Der Kommentar wird als Block-Kommentar eingefügt, wobei ein Stern mehr verwendet wird, als für den eigentlichen PHP-Kommentar notwendig ist (/**). Die so erstellten speziellen Kommentare werden als DocBlocks bezeichnet. Die wichtigsten Meta-Befehle der DocBlocks sind in Tabelle 5.1 zusammengefasst.

Meta-Befehl	Bedeutung
@author	der Autor des folgenden Codeabschnitts
@version	die Version des folgenden Codeabschnitts
@since	eine Versionsnummer oder ein Datum
@copyright	beispielsweise Name und Datum oder Firmenname
@todo	beschreibt eine noch zu erledigende Aufgabe
@link	ein weiterführender Homepage-Link
@param	die Parameter (Wert und Typ) der Methode in der Reihenfolge der Angabe bzw. der Übergabe
@var	beschreibt eine Eigenschaft bzw. Variable
@return	der Typ des Rückgabewerts der Methode

Tabelle 5.1: Einige Schlüsselworte für die Kommentierung mit PHPDocumentators

Das Werkzeug PHPDocumentator interpretiert nun die Kommentare und erstellt neue Dokumente, die den Quellcode dokumentieren. Die Dokumentation kann erstellt werden

- in verschiedenen HTML-Versionen
- im PDF-Format

- im Windows-Helpfile-Format (CHM)
- als Docbook XML

Die HTML-Versionen der Dokumentation sind dabei am weitesten verbreitet. Der Vorteil liegt darin, dass auf Verweise zu anderen Klassen und deren Methoden über Hyperlinks zugegriffen werden kann. Dadurch entsteht ein Mehrwert im Vergleich zu einer rein textuellen Dokumentation.

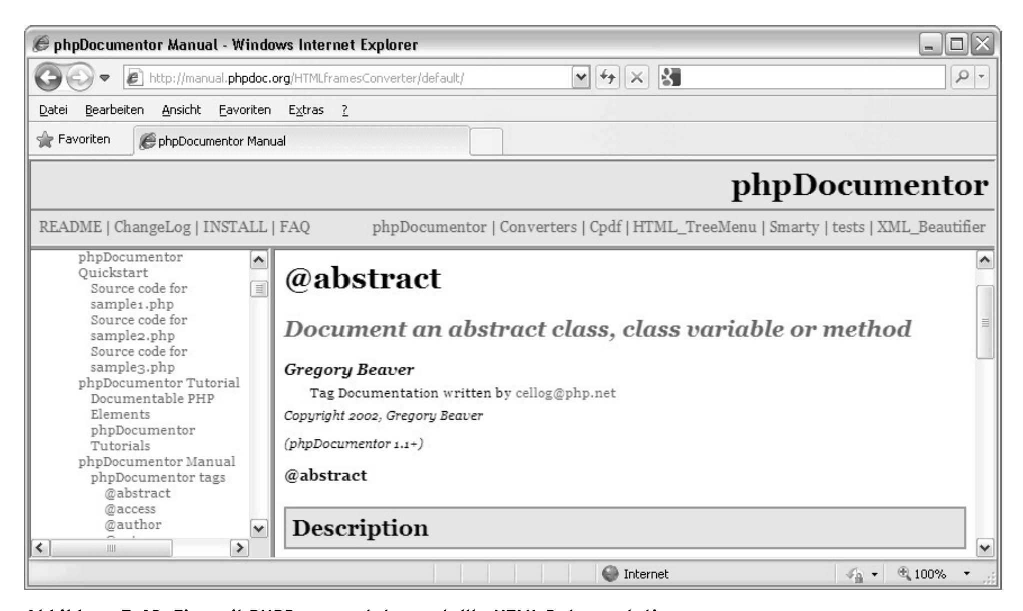


Abbildung 5.49: Eine mit PHPDocumentator erstellte HTML-Dokumentation

Fehler finden mit XDebug

Beim Erstellen Ihrer ersten PHP-Klassen ist Ihnen wahrscheinlich bereits aufgefallen, dass Fehler im Quellcode oft nur schwer erkannt werden können. Der Befehl *var_dump($x)* ist zwar hilfreich, die Ausgabe eines einzelnen, komplexen Objekts *$x* kann jedoch bereits sehr unübersichtlich sein.

Tritt ein Fehler auf, so erhält man im Internetbrowser des Clients oft keinerlei Ausgabe. Die Frage, in welcher Methode gerade ein Fehler aufgetreten ist, lässt sich bislang nur anhand von Statusausgaben beispielsweise in einer Logdatei realisieren. Gerade bei komplexeren Anwendungen ist dies unzureichend.

Um diesen Zustand zu verbessern, wurde die PHP-Erweiterung XDebug entwickelt, die als Open-Source-Anwendung sowohl für MS Windows als auch für Linux verfügbar ist. Wie bei PHPUnit bereits beschrieben, wird auch XDebug über den PEAR-Installer in den entsprechenden WAMP- bzw. LAMP-Server integriert. Dabei bietet XDebug im Wesentlichen zusätzlich zu PHP:

- eine verbesserte *var_dump*-Ausgabe
- eine Beschränkung der Rekursionstiefe bei Methodenaufrufen

- eine Protokollierung der aufgerufenen Methoden incl. deren Ein- und Ausgabeparametern (Stack Trace)
- eine Messung des Laufzeitverhaltens und des benötigten Speichers von PHP-Skripten auf dem Server (Profiling)
- eine Analyse der Codeabdeckung (Code Coverage), um zu prüfen, ob nie genutzte Codeteile existieren
- die Möglichkeit, eine interaktive Fehleranalyse unter Verwendung von Haltepunkten und einer Einzelschrittausführung von PHP-Skripten durchzuführen

Abbildung 5.50 zeigt die XDebug-Ausgabe eines Skript-Tracings. Entwickeln Sie in einer PHP-Entwicklungsumgebung, so können diese Ausgaben meist in einer besseren grafischen Formatierung dargestellt werden.

```
TRACE START [2007-01-22 08:06:31]
    0.0002      33312   -> {main}() /lib/core/admin-parser.php:0
    0.0004      33344     -> error_reporting() /lib/core/admin-parser.php
    0.0005      33376     -> ob_start() /lib/core/admin-parser.php:27
    0.0007      75256     -> include_once(/lib/core/functions/init.inc) /
    0.0008      75256       -> dirname() /lib/core/functions/init.inc:30
    0.0009      75720       -> require_once(/lib/conf/defines.inc) /lib/c
    0.0009      75752         -> define() /lib/conf/defines.inc:32
[...]
    0.0485     133456     -> ob_get_clean() /lib/core/admin-parser.php:41
    0.0493      97376   -> Core_Init->__destruct() /lib/core/classes/clas
    0.0494      97376     -> trace_dump_xdebug() /lib/core/classes/class.
    0.0494      97376       -> function_exists() /lib/core/functions/debu
    0.0494      97400       -> xdebug_get_tracefile_name() /lib/core/func
    0.0495      97432       -> is_file() /lib/core/functions/debug.inc:11
    0.0495      97464       -> xdebug_stop_trace() /lib/core/functions/de
    0.0495      97464
TRACE END   [2007-01-22 08:06:31]

ISO8-----XEmacs: trace.2043925204-2.xt      (Inferior GDB Frame:no proces
```

Abbildung 5.50: Konsolenausgabe von XDebug

Der Editor PHPedit

Bei PHPedit in der aktuellen Version 3.4.4 handelt es sich um einen Editor für PHP-Code, der neben Syntax-Highlighting auch eine Projektverwaltung, eine automatische Vervollständigung von Befehlen und Debug-Tools beinhaltet.

Neben der Dokumentation mit PHPDocumentator können auch SVN und PHPUnit in den Editor integriert werden. PHPedit verfügt außerdem über ein FTP-Tool, mit dem Sie die erstellten Skripte automatisch auf einem Server hochladen können.

PHPedit ist für Windows-Plattformen ausgelegt und benötigt das Microsoft-.NET-Framework 2.0. Von der Homepage *http://www.phpedit.com/* können Sie eine kostenlose 30-tägige Testversion herunterladen. Neben einer kostenlosen Studentenversion als „Personal License“ zu Lernzwecken existiert eine Professional-Lizenz für Unternehmen, die 89 € kostet. Diese Lizenz beinhaltet aber leider nur den reinen Editor. Für die Nutzung von FTP, SVN und PHPUnit müssen Sie die Premium-Lizenz erwerben, die 179 € kostet.

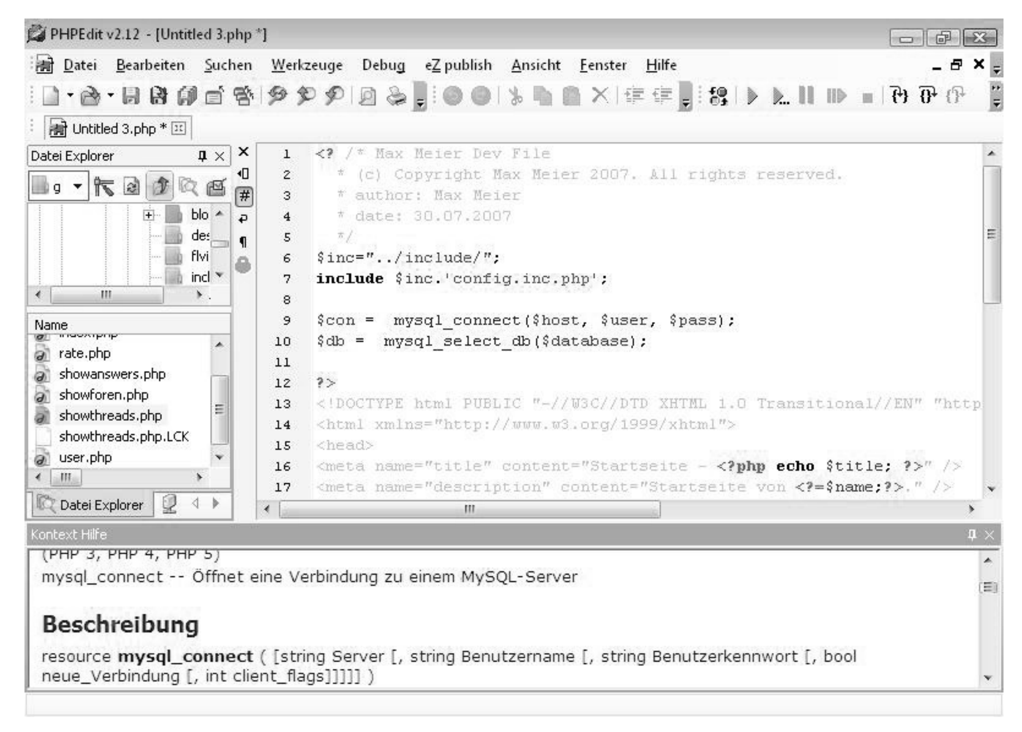


Abbildung 5.51: Der Quellcodeeditor PHPEdit

Der Editor UltraEdit

Der für Windows und neuerdings für Linux erhältliche Editor UltraEdit existiert bereits in der 15. Version. Im Gegensatz zu PHPedit ist UltraEdit nicht auf die Sprache PHP beschränkt und unterstützt zusätzlich eine Syntaxhervorhebung für C/C++, VisualBasic, HTML, Java und Perl mit Optionen für Fortran und LaTex. Andererseits ist die Abstimmung auf PHP natürlich nicht so weit fortgeschritten wie bei PHPedit. So fehlt eine feste Integration von Werkzeugen wie PHPUnit, SVN oder PHPDocumentator. Stattdessen verfügt UltraEdit über eine Werkzeugkonfiguration, mit der Anwendungen auf der Kommandozeile durch einen Mausklick oder über eine Tastenkombination angestoßen werden können, deren Ausgabe in den Editor umgeleitet werden kann. Auf diese Weise können andere Werkzeuge integriert werden.

UltraEdit bietet Unicode-Zeichensatzunterstützung, eine Rechtschreibprüfung für mehrere Sprachen, einen integrierten FTP-Client, eine konfigurierbare Tastenbelegung, einen Hex-Editor sowie eine HTML-Werkzeugleiste.

Unter *http://www.ultraedit-germany.de/* finden Sie weitere Informationen zu dem Editor, dessen Einzellizenz ca. 50 € kostet.

```
<?php
for($i=0;$i<$_SESSION[BestandAnzahl];$i++){
  $bestand=unserialize($_SESSION[Bestand][$i]);
  $bilanz=$bestand->getBilanz();

  echo '<font face="Arial,Helvetica">';
  echo '<b>'.$bilanz[Aktienname].'</b>';
  echo ' ('.$bilanz[AktienISIN].')</b>';
  if($bestand->isVerkauft()){
    echo ' &lt;&lt;VERKAUFT&gt;&gt;<br>';
  }
  else{
    echo ' <a href="uebersicht.php?w='.$i.'">wählen...</a><br>';
  }

  echo $bilanz[anzahl].' Stück am '.$bilanz[ekDatum].' gekauft';
  if($bestand->isVerkauft()){
    echo ' und '.number_format($bilanz[tageImBesitz],0).' Tage gehalten';
  }
  else{
```

Abbildung 5.52: Der Quellcodeeditor UltraEdit in der älteren Version 9.00c

Die Entwicklungsumgebung Eclipse für PHP

Während es sich bei PHPedit und UltraEdit in erster Linie um reine Quellcodeeditoren handelt, sind Eclipse für PHP und das im folgenden Kapitel vorgestellte Zend Studio Entwicklungsumgebungen, mit denen insbesondere größere Projekte verwaltet werden können. Bei Eclipse handelt es sich um ein Open-Source-Projekt, das selbst in Java programmiert wurde. Es basiert auf einem OSGi-Framework namens Equinox und stellt eine hochmodulare Plattform dar, die aus einem minimalen Kern besteht, auf den zahlreiche Plug-ins installiert werden können.

Die integrierte Entwicklungsumgebung wurde ursprünglich nur für Java-Anwendungen verwendet und ist im Java-Umfeld sehr weit verbreitet. Die Entwicklungsumgebung selbst unterstützt jedoch nicht nur eine einzige Programmiersprache. Das unter Windows, Linux und Mac einsetzbare Werkzeug kann auf der Homepage *http://www.eclipse.org/* im Downloadbereich auch in einer C/C++-Version und eben auch für PHP-Entwickler heruntergeladen werden. Die Version „Eclipse for PHP Developers“ ist 139MB groß.

Die Integration der Sprache PHP ist in der Community bislang noch nicht so weit ausgeprägt wie im Java-Umfeld. Dennoch lassen sich die meisten Hilfsprogramme wie Unit-Testing, Debugging oder Versionierung als zusätzliche Plug-ins in die Entwicklungsumgebung integrieren.

Eclipse for PHP Developers integriert die so genannten Eclipse PHP Development Tools (PDT), die Syntax Highlighting und weitere grafische Hilfsmittel für den Entwickler mit-

bringen. So existiert beispielsweise eine grafische Symbolik für private und öffentliche Eigenschaften bzw. Methoden und ein weiteres Symbol, das das Überschreiben einer Methode von einer Oberklasse kennzeichnet.

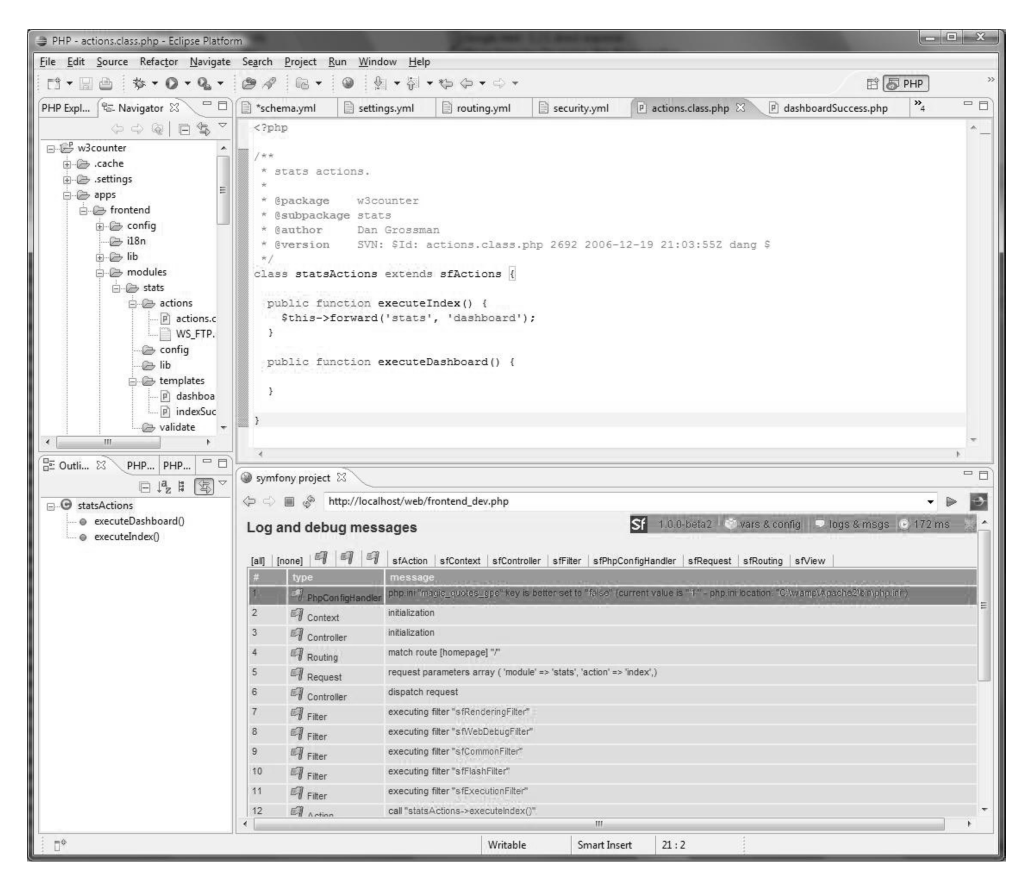


Abbildung 5.53: Screenshot der Eclipse-Entwicklungsumgebung für PHP

Die Entwicklungsumgebung Zend Studio

Das Unternehmen Zend Technologies ist seit Jahren ein führender Anbieter von PHP-Produkten und Dienstleistungen für die Entwicklung, Implementierung und Verwaltung von PHP-Anwendungen.

Zend beschreibt seine Entwicklungsumgebung Zend Studio in der aktuellen Version 7.0 als ein Integrated Development Environment (IDE) für professionelle Entwickler, in der alle Entwicklungskomponenten enthalten sind, die für den gesamten Lebenszyklus von PHP-Anwendungen benötigt werden.

Ebenso wie Eclipse integriert auch Zend Studio die Eclipse PDT, bietet jedoch im Vergleich zu Eclipse for PHP Developers eine tiefere Integration der Sprache PHP. Eine Ursache liegt darin, dass sich die Eclipse Foundation in erster Linie auf die Sprache Java konzentriert, während Zend Studio ausschließlich für PHP ausgelegt ist. Ein weiterer

Grund liegt darin, dass es sich bei dem Zend Studio nicht um ein Open-Source-Projekt handelt und die Weiterentwicklung der Entwicklungsumgebung aus Lizenzeinnahmen möglich ist.

Eine Einzellizenz kostet ca. 400 € und umfasst auch ein 1-Jahres-Abonnement für Updates und Support. Der Download einer Testversion von der Homepage *http://www.zend.com/en/products/studio/* ist möglich, um ein erstes Gefühl für die Entwicklungsumgebung zu erhalten. Wie auch Eclipse ist Zend Studio sowohl auf Microsoft Windows als auch auf Linux und Macintosh-Betriebssystemen funktionsfähig. Auf der Homepage *http://www.zend.com/en/products/studio/comparison* finden Sie eine Gegenüberstellung von PDT und Zend Studio 7.0 mit einer Auflistung der Mehrwerte, die Zend Studio 7.0 gegenüber PDT zu bieten hat.

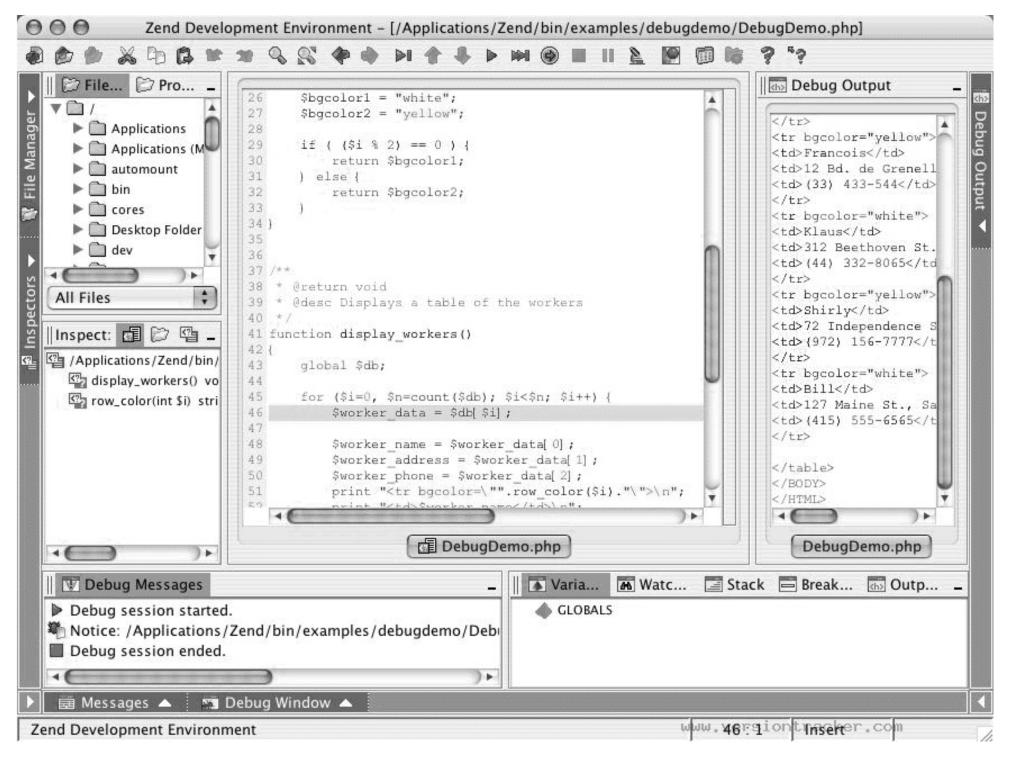


Abbildung 5.54: Screenshot der Entwicklungsumgebung Zend Studio

> **Hinweis**
>
> In diesem Kapitel wurden lediglich einige wichtige Werkzeuge im PHP-Umfeld skizziert, damit Sie sich einen ersten Überblick verschaffen können. Für einen tieferen Einblick werden Sie sich detaillierter mit den für Sie interessanten Werkzeugen beschäftigen müssen. So werden beispielsweise PHPUnit, Subversion SVN, XDebug und PHPDocumentator in dem Buch mit dem Titel „Enterprise PHP Tools" von Stefan Priebsch, ISBN 3-9350-4293-0 des Verlags entwickler.press genauer beschrieben.

Stichwortverzeichnis

Symbole

Numerisch

A

B

C

D

M

N

O

P

Q

R

S